Guide to ARMLinux

for Developers

Aleph One Ltd.

Wookey

Paul Webb

Guide to ARMLinux: for Developers

by Aleph One Ltd. Wookey, and Paul Webb

Published 2002.05.03

This guide is for developers working with GNU/Linux on ARM devices. Detailed set-up instructions are given for the Delft Technical University LART, the Intel Assabet/Neponset boards, and the Psion Series 5. However you can also use the book to help ARMLinux development on any ARM device. A reasonable level of technical expertise is assumed, and familiarity with Linux is helpful but not necessary.

Table of Contents

List of Tables

List of Figures

Preface

Welcome!

We hope you find this book useful. If you have bought it along with another Aleph One Product such as the LART development kit, or Aleph ARMLinux, then please take the time to read it so that you know what you have, where to start, how to install, and where to turn if you have problems.

1. Credits

ARMLinux for Developers would not have been possible without the work of many people. We would therefore like to particularly thank:

- Jan-Derk Bakker and Eric Mouw of the LART project for LART, Blob, help and vision;
- Remote 12 Systems of London for LART part sourcing;
- Chris Rutter for the GNU Toolchain chapter, the Debian Autobuilder and the http://www.armlinux.org/ facilities;
- Brian Bray for the *ARM Structured Alignment FAQ*;
- John G. Dorsey for his description of installing ARMLinux on an Intel Assabet;
- Russell King for the original ARM kernel port, continuing maintenance, and the http://www.arm.linux.org.uk/;
- Stephen Harris, George Wright and Nathan Catlow for their Psion documentation;
- Phil Blundell and Jeff Sutherland for work on and advice about the ARM Toolchain, and Phil for work on Debian and Bootfloppies;
- Nicolas Pitre for his work on the kernel, and the JFlash utility;
- Steve Wiseman for his work on JTAG;
- Holly Gates for the JTAG dongle design;
- Nicolia Mahncke for LART power supply info;
- Abraham van der Merwe for LART LCD display info;
- and of course the many others who have worked on Debian ARM, the ARM kernel port and the people whose expertise we have tried to incorporate into this book.

We also acknowledge all Trademarks used in this book.

2. Outline of the Book

Although it is possible to read this book from beginning to end, most users will want to just read the most relevant sections. Everyone should read the introduction and Chapter 2. If you have bought it for use with a particular item of hardware then read the appropriate chapter - you will be referred to other parts of the text for more detail should you need it. Those who are not familiar with desktop Linux should read Chapter 10 to get a feel for how to use it, before trying to install anything. Developers working with a new device or with a general interest will find the chapters on tools and techniques most useful.

3. Buying the Book

Although this guide is available in HTML and PDF formats to all users who buy the Aleph ARMLinux CDs, a printed version of the Guide can also be bought separately from Aleph One Limited.

Interested users should go to http://www.aleph1.co.uk/armlinux/thebook.html or <info@aleph1.co.uk> for further details.

4. Dedication

Chris Rutter was killed by a car whilst crossing the road during the preparation of this book. His enthusiasm was instrumental in my starting this project so it seems appropriate to dedicate this book to him.

Chapter 1. Introduction

This book tries to cover a range of hardware and aspects of ARMLinux. It assumes that you are at least a competent computer user, more likely an experienced developer, but perhaps not very familiar with GNU/Linux or the ARM and its development platforms.

There are many things that are specific to different items of hardware, and many things that are common across various devices. We have tried to avoid too much repetition, but we have also tried to make the text reasonably linear in the hardware chapters. This is inevitably a compromise. In general you should find that reading through the relevant hardware chapter will get you going, but will not go into much depth about why you are doing things and what other options there might be at each stage. We refer to other chapters that have more details on each aspect (using JFlash, patching the kernel, etc) throughout these texts. If you have feedback on the book we'd be very happy to have it, so as to improve future versions - see Chapter 14.

In a fast-moving field like this, this book will always be a work-in-progress. We give the current state of the art at the time of writing, but recognise that this will soon change. Thus we try to provide links to the places online where you can get the latest info. When you find things that are out of date, or just plain wrong, please tell us.

There will be future editions covering more devices and more subject areas. Things that will definitely be in the next release are coverage of porting the kernel to a new ARM device, more information for developers using a Windows host PC rather than a Linux one, and more on debugging and simulation techniques.

1.1. The Aleph ARMLinux CD

If you have bought this book as part of our Aleph ARMLinux distribution then be aware that this follows the hallowed GNU/Linux principle of *releasing early and often*. The distribution is not yet finely tuned to the needs of various embedded devices. It has parts that are specific to small ARM devices, and parts which are generic, but assume a desktop environment. The specific part contains this document, kernels, utilities and RAMdisks suitable for the devices covered in the book, pre-compiled x86/ARM toolchains, and current kernel source and ARM patches. The generic part forms the bulk of the distribution and is essentially the standard Debian ARM distribution, containing a huge array of packages and all the corresponding sources. These are very useful, providing an enormous library of things that you can run on your ARM hardware, but the Debian installer is not yet set up to install to embedded systems. You need to make your own RAMdisks from the parts available to use these binaries. There are several projects underway to

make this process easier, including emdebsys from http://www.emdebian.org/ which is included. Future releases will have significant improvements in this area.

Have fun!

1.2. Nomenclature

Throughout this book we distinguish between the Linux kernel and the GNU/Linux operating system. Many people call both of these things *Linux*, but this is somewhat misleading and certainly unfair to the GNU project which has created almost all of the basic operating system software (shells, compilers, editors, tools, documentation systems, document processors etc). It had done all this before the Linux kernel came on the scene and enabled the powerful combination we see today of GNU software running on the Linux kernel. We feel the distinction is important as well as reducing confusion and thus maintain it in this document.

Chapter 2. Embedded ARMLinux: Fundamentals

2.1. Introduction

The fundamental requirement for a GNU/Linux system is a kernel and a filing system for it to execute things from.

The kernel is an autonomous piece of code that doesn't need any other files or libraries to get started, although without a root filing system it will simply stop after initialising the system as it has no files to operate on.

The filing system can be loaded from a range of different hardware devices (hard disk, RAM, ROM, CD and NFS mount) and must be in a format that the kernel understands. For embedded systems it is often a RAMdisk. The kernel and RAMdisk both need to be loaded to correct places in memory and the kernel executed. This can be done in various ways, typically by loading over a serial connection or from local flash RAM.

In order to be able to load anything from anywhere some kind of boot loader must be present on the target hardware which knows how to load files and execute them. This can be installed in a physical fashion by inserting a pre-programmed ROM/EEPROM or flash chip, but is more usually installed using a JTAG port, which allows instructions to be executed from an external input, and thus an initial program loaded.

As there are so many possible boot scenarios we cannot describe them all in detail. We will describe only the most common (which are the most useful for typical situations) but also make the principles clear so you can do something more exotic if you need to.

2.2. Boot loaders

Boot loaders are highly system-specific, but given the huge number of ARM-based systems, and a much smaller number of boot loaders there are now several modularised loaders which easily support multiple platforms, as well as many loaders that are specific to one platform, or several very similar ones. This can be a boot ROM like the NeTTrom in a NetWinder, or it can be the native OS like RISC OS in a Risc PC, or EPOC in a Psion5, followed by a native OS application that loads ARMLinux, or it can be a dedicated boot loader installed into Flash, like Blob, Angel, or Bootldr in the LART, Assabet and iPAQ. If you are porting to a new

ARM platform then you will need a boot loader. You may well be able to base this on one of the existing open source ones, or you may already have something that can be modified to load ARMLinux instead of whatever the device used before.

There are two boot loaders commonly used with the LART and Assabet: Blob and Angel. Angel is part of ARM's Development toolkit, and can also be used as a debugger, but we use it with a companion program Angelboot instead. Angel is a simple loader that receives a kernel and RAMdisk over the serial port and loads them to specified addresses before running the kernel. Blob is a more complex program that can do the same as Angel but can also install a kernel and RAMdisk permanently in flash RAM so that the target can subsequently boot autonomously.

See Chapter 6 for a full discussion of boot loaders and examples of their use. Brief typical use is covered in the install section for the relevant target device.

2.3. JTAG

Getting the target side of the boot loader installed is normally done with the Jflash utility (both Windows and Linux versions exist). This drives the JTAG interface through the printer port of the host PC. The Linux version is called JFlash-linux.

> **Note:** This program is different for each target as it depends on knowing the exact hardware of the target, so JFlash-linux for Assabet and JFlash-linux for LART are different and you need to use the right one.

Table of devices and tools

Device	JTAG	Boot loader(s)
LART	Jflash,Jflash-linux	Blob
Assabet	Jflash,JFlash-linux	Angel, blob
Psion5	-	Arlo

2.4. Host machine

If the target has sufficient IO devices (e.g. screen and keyboard) then it can be used without a host. However, during initial development a host is normally used as a console (screen and keyboard), for compiling new code and for monitoring debug output.

There are several reasons why this is practical:

- you will probably keep rebooting the target;
- it will probably crash a lot if you are doing any low-level development, or the hardware's not quite debugged yet;
- there may well not be space for the toolchain software on the target;
- you will normally be more familiar with the host environment.

As the host is usually an x86 PC rather than another ARM based machine this means that you will need to cross-compile software to run on the target processor. You can arrange to use an ARM machine as the host (e.g. an Acorn Risc PC, NetWinder, CATS board) but even here you usually need to separate the host and target environments as they will probably be using different kernel versions and perhaps library versions.

You can have either a GNU/Linux or a Windows box as the host. As you are working with Linux on the target it is useful in many ways to also have a Linux host - the toolchain is easier to set up, you can easily mount and try out your RAMdisks etc. Much of this book assumes that you have access to a GNU/Linux host. However we recognise that many developers will have a Windows box on their desks and may not wish or be able to change just yet. The best solution is to use the Cygnus Unix-like environment for Windows which gives you the functionality you need. A chapter for this book, describing the necessary software and techniques, is in preparation for the next edition. In the meantime either use your skill and judgement, or get a GNU/Linux box too.

Getting a cross-compiler toolchain set up used to be very difficult, requiring much gurudom and attention to a host of details but with the precompiled toolchains on this CD it is very easy. Chapter 8 gives details of both the ready built toolchains you can install, and explains how to build a new one should you need something different from that which is provided.

2.5. RAMdisks

The RAMdisk is a very useful kernel facility that lets you load the files you need on the system into RAM along with the kernel. It takes the form of a compressed filesystem. The kernel automatically allocates RAM for it and uncompresses it, then mounts it as the root filing system.

Again, there are many variations possible on this theme, but the conventional setup is to format the RAMdisk as ext2, the normal Linux disk filesystem. For this to work you need to specify the correct kernel options: RAMdisk support (CONFIG_BLK_DEV_RAM, CONFIG_BLK_DEV_RAM_SIZE, CONFIG_BLK_DEV_INITRD) and support for the filesystem used (normally ext2 - CONFIG_EXT2_FS).

There are suitable example RAMdisks provided for the devices covered in this manual on the CD. There are also several available online, where developers have made their own available for others to use. We recommend that you use one of these to get started, however for many applications you will need to make your own. It’s not difficult, and the process is described in Section 7.3.

Chapter 3. LART

This chapter introduces the reader to the Linux Advanced Radio Terminal (LART) by describing hardware specifications and a range of relevant resources.

3.1. Background

The LART is part of a project at the Technical University of Delft to do wearable computer research. The first aspect of this was building a suitable platform as nothing with the right combination of low-power, high-performance and low cost existed at the time. The result is offically the ’Linux Advanced Radio Terminal’ but a look at the Canonical Abbreviation/Acronym List (http://www.astro.umd.edu/~marshall/abbrev.html) suggests this may be a contrived acronym.

Part of the design philosophy was that it could be built without expensive facilities and it was easy to add bits of hardware for your own purposes. This combined with its power/performance ratio makes it an excellent development tool. In practice, buying the parts to make them yourself is a time-consuming and difficult process, so Aleph One and Remote12 Systems have made it available as an assembled tested development kit. We hope this document will get you going in LART/ARMLinux development quickly and reasonably painlessly.

3.2. Hardware

3.2.1. Aleph One LART development kit

Many users of this chapter will have bought the Aleph One LART development kit. If you haven’t, the information here will still be useful but references to the supplied connectors and cables will need to be adjusted to whatever equipment you are using. The Aleph One kit comprises:

- LART main board with Blob boot loader installed;
- this Guide to *ARMLinux for Developers* on paper and CD-ROM;
- the Aleph ARMLinux distribution for LART including suitable kernels, RAMdisks and tools for the LART, as well as all the Debian 2.2release2

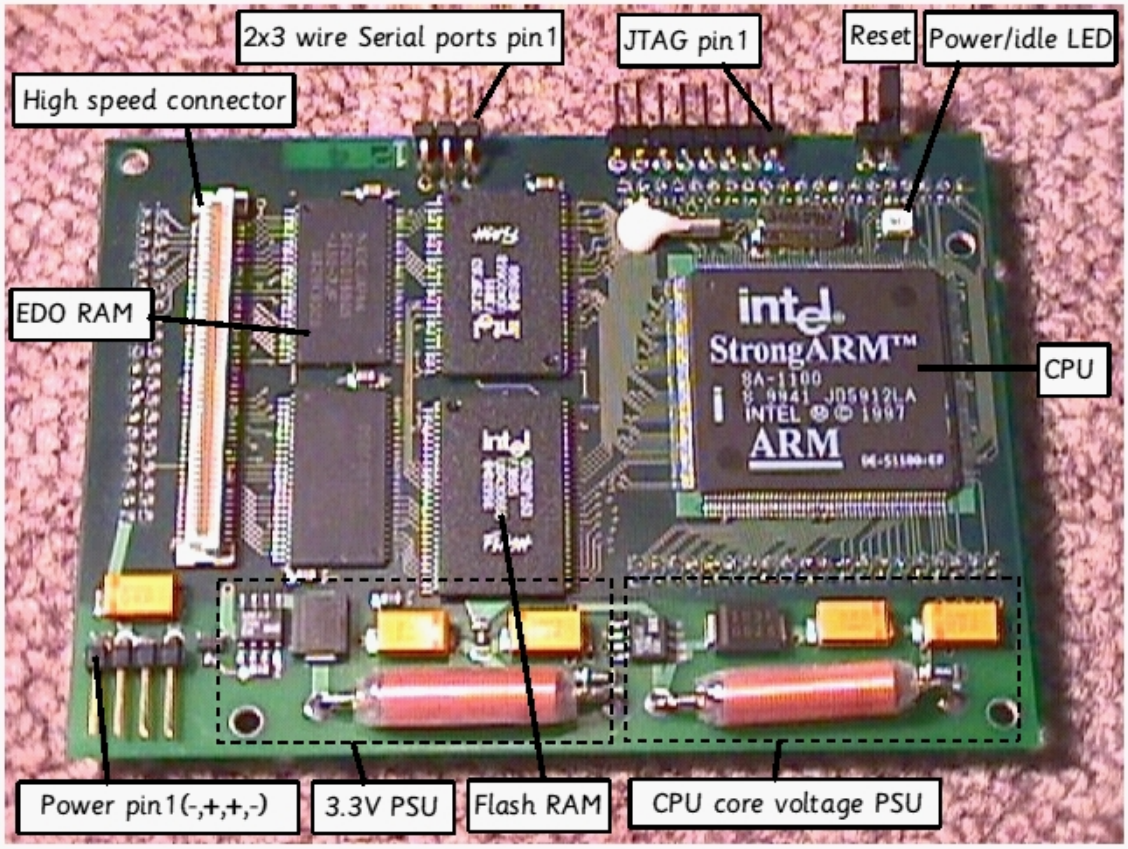
packages in binary and source form on 5 CDs;

- JTAG adaptor board;
- parallel extension cable;
- twin-headed serial cable;
- power cable.

This is everything you need to get going (apart from a host machine); we hope you enjoy developing on it.

Up to date information is also available from http://www.aleph1.co.uk/armlinux/LART/

3.2.2. Mainboard



LART mainboard (showing top side)

The specifications for the LART mainboard are as follows:

- 220 MHz Digital SA-1100 StrongARM CPU;
- 32 MB EDO RAM;

- 4 MB Intel Fast boot block Flash memory;
- power usage of less than 1W;
- performance in excess of 200 MIPS.

The board can run standalone, booting an OS from Flash.

The 4MB Flash is sufficient for a:

- boot loader;
- compressed kernel;
- compressed RAMdisk.

The LART accepts an input voltage of between 3.5 and 10 Volts, although it can be modified to accept up to 16V. The on-board DC-DC converters have an efficiency of between 90 and 95%.

3.2.3. Connectors

3.2.3.1. Signal

Almost all signals from the SA-1100 are available on the external connectors.

The High-speed Connector is an SMD with 0.8mm pitch and offers:

- access to the 32-bit data bus;
- access to all 26 address lines;
- high-speed peripherals with a data rate in excess of 400 MB/second.

The Low-speed Connector is actually three separate through-hole connectors with a 2mm pitch which:

- export General Purpose (GP) I/O pins and most of the data/address buses to implement ISA or PCMCIA-based peripherals;
- are ideal for connecting to low-speed devices.

3.2.3.2. Serial

The LART serial connector provides:

- two simple serial ports with no handshaking.

3.2.3.3. Other

Additional connectors include a:

- reset connector;
- voltage/current measurement point;
- power connector;
- JTAG connector.

3.2.4. Kitchen Sink Board

The Kitchen Sink Board (KSB) provides the LART with a:

- stereo 16-bit 44k1 audio output at line and headphone levels;
- 44-pin 2mm IDE/ATA interface;
- connector for a single or quad Ethernet board;
- two PS/2 connectors for keyboard and mouse;
- mono audio I/O from an UCB1200;
- connectors for POTS, IrDA, USB clients, video and touchscreens.

3.2.5. Ethernet Boards

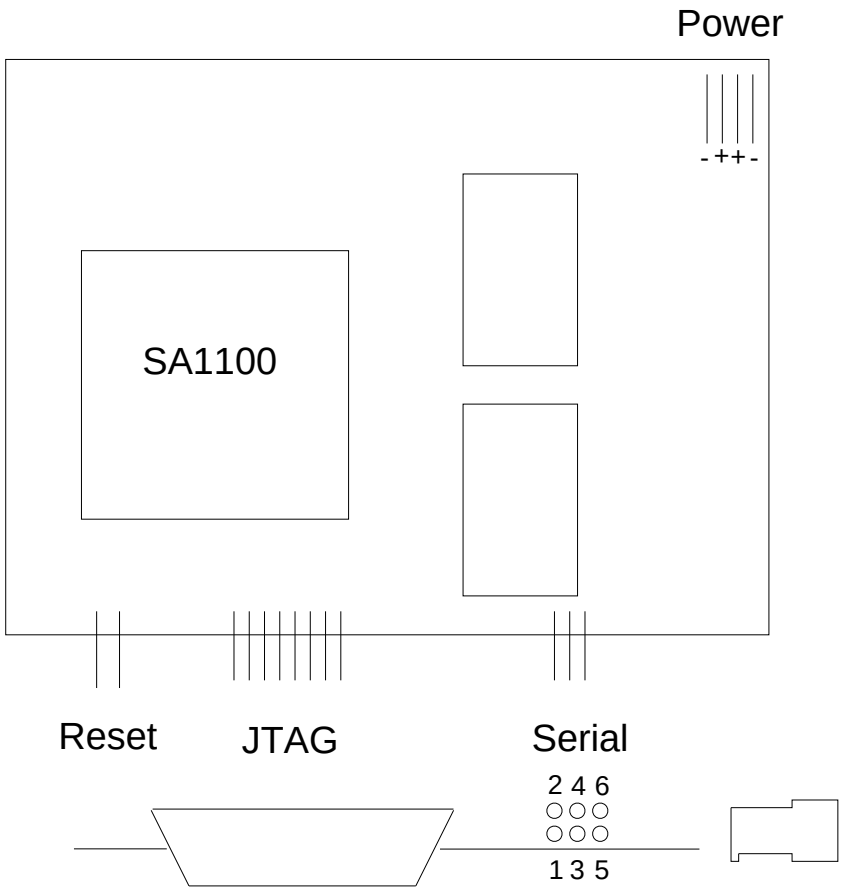
The *Ether1* board is an ethernet adaptor that connects the LART to an ethernet through the Kitchen Sink Board. The core of the board is a Crystal CS8900A Ethernet chip (http://www.cirrus.com/design/products/overview/index.cfm?ProductID=46).The other necessary part is the *Ethcon* board which is a 10-base-T adaptor that goes between the ethernet board and the UTP cable. This has various physical arrangement possibilities which will be described below.

A 4-channel ethernet design, useful for routers, also exists for the LART but has not been built by Aleph One. The schematic is on the CD.

3.3. Installing for the LART

3.3.1. Power and Serial Connections

Refer to the diagram and picture to familiarise yourself with the device. The view showing the StrongARM CPU and large inductors is the 'top' of the board throughout this document.



Layout of LART showing connectors

Power is supplied to the four pin connector in the top right. The supplied power lead is wired such that it doesn't matter which way round you plug it in - polarity will be correct. (The two central pins are +ve, the two outer -ve). The red LED next to the crystals in the lower left hand corner should glow when power is applied. At 5V, initial start-up current for a lone LART is ~160mA, dropping to ~80mA after a few seconds. The SMPS design means there is a significant surge on start-up where up to 1A is drawn during the first second of operation. The power supply needs to be able to supply at least 400mA during this surge for the SMPS to start up properly. So if you are using a bench supply make sure it is set to supply enough current.

Warning

The supply voltage is between 4V and 10V. Some documents (including the LART web site at the time of writing) give an upper limit of 16V but this will damage your LART! The LT1266 can take 20V nominal, but the design uses voltage doubling so the input voltage must remain below 10V unless the PSU is changed around (which can be quite easily done).

The serial connection is the 2x3 header. This supplies two simple serial ports (ground, data out, data in). The supplied cable should be plugged in so that the lip on the blue 2x3 connector points downwards. The white 9-pin-D connector is serial port 1; the blue 9-pin-D connector is serial port 2. Plug the white connector into your host computer.

If you need to re-flash the Blob loader then you need to attach the JTAG dongle to the 1x8 header. The JTAG board should be attached so that the components on it are on top - i.e. components are on the same side as the StrongARM CPU on the LART.

3.3.1.1. Configuring the LART for input voltages between 10-16V

As supplied, the LTC1266 switcher is configured for maximum efficiency by using an N-channel rather than a P-channel MOSFET. The input voltage (Vin) is doubled to drive the N-channel MOSFET and as the LTC1266 is rated for a maximum of 20V, that means that Vin cannot be more than 10V. Should you wish to use a higher input voltage (e.g. 12-14V from a vehicle) then you need to re-arrange things to remove the voltage doubler.

With the modifications described below, the Vcore regulator will use 3v3 as its gate drive voltage. This limits Vin to 16.7V maximum. The added advantage of this is that gate drive dissipation is fixed rather than increasing with Vin. Depending on input voltage you might gain from 0% to 10% efficiency improvement over a span from 4V to 16V.

The 3v3 regulator will start up with 3.0V gate drive from the linear regulator. As the 3v3 output rises past 3.0V the gate drive will come from its own output due to the OR'ing of the two diodes. This fixes 3v3 gate drive dissipation as with Vcore. The linear regulator should only draw its own quiscent current after startup.

The above has been tested on an actual LART rev.4 board. The Vcore fix is not tested but (should be) guaranteed by design.

If you get trouble with noise on 3v3 insert a 10 Ohm resistor in series with each of the anodes of D2, D5 AND decouple the anodes to ground with at least 1μF ceramic/10μF tantalum. The more μF the better.

Detail of circuit changes

- Change C50, C51 to at least 20V types. AVX recommends they be rated at double working voltage especially with low source impedances like NiCads and hot plugged supplies, and they mean it. When the 16V rated fails it will char an inch wide area of the board halfway through.
- Change T1, T2 to 30V types with 20Vgs-max and reasonable (>1.5Amp) Ids @ 2.7V gate drive.
- Lift the anode of D5 and connect it to +3v3 on C57.
- Make a small low drop linear 3.0V regulator with at least 1μF ceramic/10μF tantalum output capacitance the more the better. Preferably one with good transient response. It needs to be rated at 50mA or better at 20Vin. Farnell have some from National Semiconductor.
- Connect the regulator input to Vin and the output to the lifted anode of D2.
- Connect the cathode of a second ZHCS750 to D2's cathode. Connect its anode to +3v3 on C57. A dual schottky can be used but must be rated for 30V and >200mA peak.

3.3.2. Using JTAG and JFlash-linux

It is possible to program flash memory through the SA-1100 JTAG interface. A utility to do this is provided. Aleph One supplied LARTs will already have had this done so you can skip to the next section.

In order to use the JTAG programmer, take the following steps:

- connect the parallel cable to your host computer's printer port;
- connect the JTAG dongle to the cable;
- connect the other end of the JTAG dongle to the LART, with the JTAG component side the same way up as the LART main board;
- turn the LART on.

You can then issue the following command as root:

./Jflash-linux precompiled-blob-2.04

and you should see something like the following after which the LART will reset itself before running Blob.

```
using printer port at 378
```

```
Seems to be a pair of 28F160F3, bottom boot. Good.

Starting erase for    da5 bytes
Erasing block   0
Erasing done
Starting programming
Writing flash at hex address      1b0, 12.37% done
Writing flash at hex address      3d0, 27.94% done
....
Programming done
Starting verify
Verifying flash at hex address      4e0, 35.73% done
....
Verification successful!
```

3.3.2.1. LART JTAG reset problem

Output like this is typical of the 'JTAG reset problem':

```
error, failed to read device ID
ACT: 0000 0000000000000000 00000000000 0
EXP: X001 0001000010000100 00000110101 1

failed to read device ID for the SA-1100
```

This is solved by putting a 100 OHM resistor between pins 6 (+3.3V) and 7 (nTRST) on the JTAG connector. nTRST will thereby be pulled to +3.3V so the JTAG bus won't be able to reset itself. This modification is already done on Aleph One supplied JTAG boards as they don't work without it. You can make other modifications to have the same effect (e.g. connect the JTAG reset pin to the MAX811 reset chip on the LART).

3.3.3. Serial Monitor

The only means of communicating with a bare LART is via its serial port. You should run a suitable piece of terminal software on your host. For a GNU/Linux host we recommend Minicom, which is very capable and easy to use. On Windows we recommend that you avoid Hyperterm which really isn't very good - we recommend Terraterm Pro.

3.3.3.1. Configuring Minicom

In order to configure Minicom, take the following steps:

1. run it with **minicom -o** to get to setup;
2. set it to VT102, serial port ttyS0 (or ttyS1 if you are using that port) 9600baud 8N1 (8 data bits, No parity, 1 stop bit), no modem init string, no hardware handshaking;
3. save that as *lart* config. Running Minicom as **minicom lart** will run it up with these options;
4. if you don't use Minicom for anything else then you can make these settings the overall default - Minicom calls it `dfl` - settings.

If you aren't sure which serial port you are using for the device, but your mouse or modem is in the other one then look at these things to determine where the mouse or modem are plugged in:

- `/etc/gpm.conf` will show what device gpm is using as your mouse;
- **ls -l /dev/mouse** is normally a link to the serial port your mouse is plugged into if you have a serial mouse;
- **ls -l /dev/modem** will give the same info for your modem.

If you don't have anything in either port then just try both with the following boot procedure.

3.3.4. Starting up

OK, now we're ready to go.

The reset pins are at the bottom left of the board, and a jumper is supplied to do a reset. The board will remain in reset as long as the two pins are connected. You may wish to attach a switch to these pins if you are resetting a lot!

On reset you should then see (from within your serial monitor) something like:

```
Consider yourself LARTed!

Blob version 2.0.4,
Copyright (C) 1999 2000 2001 Jan-Derk Bakker and Erik Mouw
Copyright (C) 2000 Johan Pouwelse
Blob comes with ABSOLUTELY NO WARRANTY; read the GNU GPL for details.
This is free software, and you are welcome to redistribute it
under certain conditions; read the GNU GPL for details.
Memory map:
  0x00800000 @ 0xC0000000 (8 MB)
  0x00800000 @ 0xC1000000 (8 MB)
  0x00800000 @ 0xC8000000 (8 MB)
```

```
  0x00800000 @ 0xC9000000 (8 MB)
Loading blob from flash . done
Loading kernel from flash ....... done
Loading ramdisk from flash ............... done
```

> **Note:** It always says `Loading kernel/ramdisk` whether or not a kernel and RAMdisk are actually present. It will just hang at `Starting kernel ...`if it tries to boot a 'blank' kernel.

```
Autoboot in progress, press Enter to stop ...
```

If you want to install a new kernel or RAMdisk, press **Enter** within 10 seconds; this should get you a Blob prompt:

```
Autoboot aborted
Type "help" to get a list of commands
blob>
```

If you leave it it will boot normally. Aleph One LARTs are supplied with a kernel and RAMdisk installed so they can be used straight from power-up. The versions of Blob, kernel and RAMdisk that have been shipped reasonably recently are available on our web site, so you can easily get a known-working set. Earlier ones (from up to the time of CD-creation) are also included on the CD.

3.3.5. Getting a kernel and RAMdisk

There are several pre-compiled kernels available on the CD and online. When you want to compile your own refer to Section 7.2. The correct default config setting for LART is: **make lart_config**.

3.3.6. Uploading a kernel and RAMdisk

In order to upload kernels via the serial port to Blob you need to put them in the correct format. Blob expects uuencoded files. uuencode's syntax is:

uuencode {input filename} {name in uuencoded file}> output filename

so you uuencode a kernel called `zImage-2.4.6-rmk1-np2` like this:

uuencode zImage-2.4.6-rmk1-np2 zImage > zImage-2.4.6-rmk1-np2.uu

At the `blob` prompt enter **download kernel** to upload a kernel. Blob will respond with:

```
Switching to 115200 baud
you have 60 seconds to switch your terminal emulator to the same speed and
start downloading. After that blob will switch back to 9600 baud.
```

Switch to 115200 like it tells you (otherwise the upload would take hours). If using Minicom this is done by **ALT-P**, **I** for ’115200’. Then **Enter** to get back to the prompt.

You ought to be able to use Minicom’s ’ASCII upload’ at this stage, but it doesn’t seem to work reliably with no flow control, so we recommend you use a second virtual terminal and do:

cat zImage-2.4.6-rmk1-np2.uu > /dev/ttyS0, or
dd if=zImage-2.4.6-rmk1-np2.uu of=/dev/ttyS0

which simply copies the uucoded kernel to your serial port. Switch back the Minicom virtual terminal and you should see a line of dots printed as the kernel loads.

Once the upload is complete Blob will print something like:

```
(Please switch your terminal back to 9600 baud)
Received 509300 (0x0007C574) bytes
```

Switch your terminal back as directed and hit return to get a Blob prompt. You can check the size of the uploaded kernel by typing ’status’.

Now you need to do the same procedure again to upload a RAMdisk. The only difference is that the Blob command is **download ramdisk** instead, and obviously you use the filename for the (uucoded) RAMdisk in the **cat** or **dd** command.

This process is rather tedious so we’ve supplied a script which will take care of the Blob commands and terminal speed changes for you. It’s called `upload.sh`, and is on the CD in the `LART/tools` directory. You will need to set the PORT to ttyS0 or ttyS1 to suit your host machine, and set the names of the uucoded kernel and RAMdisk files you want to upload. The script assumes they are in the current directory.

Blob has the facility to save uploaded kernels and RAMdisks into its flash RAM. Enter **flash kernel** (or **flash ramdisk**) to store the uploaded file so that it becomes the default the LART will boot with on the next reset. When you are uploading a new kernel and/or RAMdisk for the first time it is usually a good idea to test it first

by just entering **boot** which will boot with the currently uploaded files. If it doesn't work then a reset will revert to the versions in flash. If it does work then upload them again and save them this time.

3.3.6.1. Which kernel/RAMdisk to use?

There are several kernels and RAMdisks supplied on the CD, and these will be updated along with the releases. We have found that a good set of software to use at the time of writing is: `blob v2.0.4`, `zImage-2.4.6-rmk1-np2`, and `ramdisk-lart-minimal-new-tty-net.gz` which is a basic RAMdisk plus support for KSB devices including IDE drives and for the ethernet card along with basic network config tools. `ramdisk-lart-minimal-new-tty.gz` is a more basic RAMdisk without the KSB device modules or network support.

3.3.7. Booting the kernel

So finally we are ready to go. At the Blob prompt enter 'boot'. It says

```
Starting kernel...
```

then if all has gone well you should see

```
Uncompressing Linux........
```

and the kernel will boot. *daa-daa!*

3.3.8. RAMDisk details

After booting you will see a login prompt. Just enter 'root' with no password to log in.

The startup scripts are in /etc/rc.d/. There is rc.hd for the IDE interface, and rc.network for the ethernet interface - you may need to change the numbers in here for your own network, although on a lot of private nets it will 'just work'. These scripts are designed to work even if the KSB or ethernet card are not present. The rc.hd script is not run by default as although it works it causes a long lieout pause whilst the driver attempts to find any harddrives present.

> **Note:** You will get what look like errors on startup saying that the network card has not been found. This is due to the way the network driver is written - it always tries to look for both eth0 and eth1 but wil not find eth1, thus giving the rather confusing error. At the time of writing you can't just remove the code

that does this as the driver stops working - the whole cs89x0.c driver needs a rewrite for embedded ARM use, which is scheduled for the kernel 2.5 series.

3.3.9. Programming the KSB programmable logic

The core of the KSB is a pair of Lattice 2064VE in-circuit programmable logic devices. These provide much of the on-board logic, and also mean that you can add your own logic to the system if you like. If you bought your KSB from Aleph One then these chips will have been pre-programmed with the current release of the logic, so you won't need to re-program them unless you need to change the logic or you built your own board. Each 2064VE has 64 macrocells (2000 gates) and runs at up to 280Mhz, with 64 IO pins (plus 4 inputs).

In principle it is possible to use the UCB1200 to program the CPLDs under the control of the LART, but the code to do this has not yet been written. Lattice provide some sample C isp-progamming code, so it should not be too difficult to implement. At the moment though, you have to use Lattice's supplied programming software which is only available for Windows.

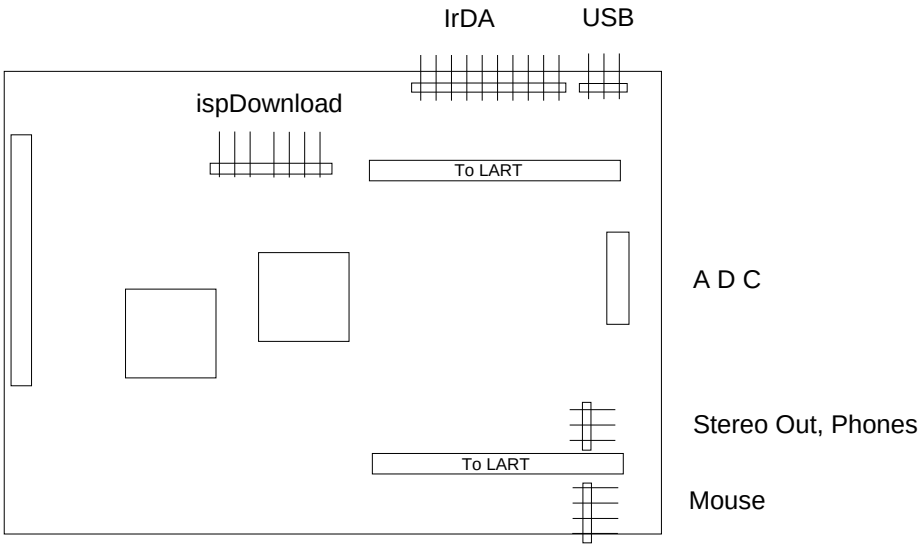
Unfortunately we are not allowed to distribute it on the CD so you have to download all 9MB of it yourself from http://www.latticesemi.com/products/destools/ispvm_system_9.cfm.

You will also need a suitable download cable, part number pDS4102-DL2. Aleph One have these in stock and you can get them from your Lattice distributor (although they tend to be in short supply), see:http://www.latticesemi.com/products/destools/proghard.cfm or you can build your own from the specs on Lattice's site: http://www.latticesemi.com/products/destools/proghard.cfm

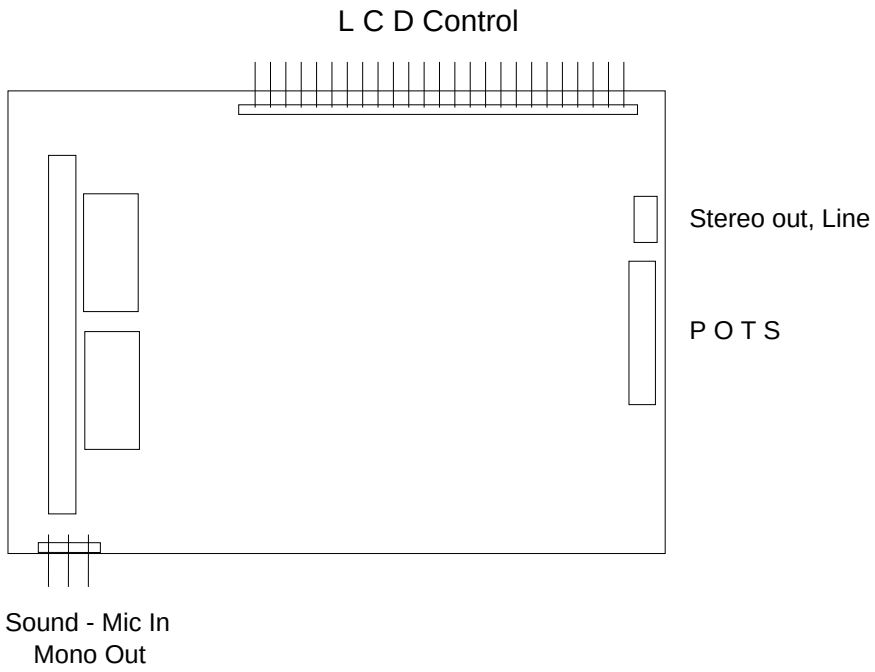
- Connect the KSB to a LART to supply it with power, then remember to power the LART. The isp software will tell you 'no board power' if you forget.
- Connect your isp download cable parts together - the RJ45 connector on the wire plugs into the blue box. The blue box plugs into the parallel port of your host machine. The 8-way connector plugs into the ispDownload connector on the KSB. The cable has pin 5 blocked so that you can't get it on the wrong way round.
- Download and install the ispVMsystem software if you haven't already. It is supplied as a self-extracting installing application so this is quite straightforward.
- Then Run the software with Start->Programs->Lattice Semiconductor->ispVM System.

- When the software starts up select ’ISP DCD’ (Daisy Chain Download), *not* ’ispVM Download’, as the latter doesn’t work for this set-up.
- The software will report that it can’t find the cable or the board is not powered if it detects these states. These messages, and programming status messages appear in the ’Messages’ window at the bottom of the ’ISP Daisy Chain Download’ window. On future occasions it will automatically load the last files programmed ready to go if the cable and board are detected.
- You should find and open the KSB.DLD file on the CD which will then display the 2 target ICs - 2064VE and the files to be programmed into them: KSB_U1.JED and KSB_U9.JED. KSB_U1 goes in the first slot, KSB_U9 in the second. The .DLD file has the paths encoded so you may well have to find the two .JED files yourself (use the browse button) if your CD does not appear as the same letter as ours. Do a ’Scan Board’ first to get the two devices detected with blank entries to put the .JED files into.
- Having got everything set up you select ’Command’->’Turbo Download’->’Run Turbo Download’ to actually program the CPLDs. Status is given in the messages window and should first list the devices as part of ’check configuration setup’, then ’Processing JTAG Chain’, ending with ’Operation done: No errors. If you immediately program another one the ’check configuration setup’ step is skipped.
- Voila - KSBs ready to go. Plug in some devices and get going.

3.3.10. KSB IO connections and devices



KSB IO connectors - top side



KSB IO connectors - bottom side

The figures show the various connectors available on the KSB. Refer to the schematics on the CD for the actual pinouts. To use the IDE interface you need to set the kernel options CONFIG_IDE, CONFIG_BLK_DEV_IDE and CONFIG_BLK_DEV_IDEDISK. Once this is done any attached IDE drivers should be recognised on boot-up. You need to add suitable device nodes like this:

mknod /dev/hda b 3 0

mknod /dev/hda1 b 3 1

mknod /dev/hda2 b 3 2

And then create a mount point for it and mount it:

mkdir /mnt

mnt -t ext2 /dev/hda1 /mnt

This assumes that the drive has already been partitioned and had the first partition formatted as ext2 on some other machine. You can do it directly on the LART if your RAMdisk has fdisk and mke2fs installed.

3.3.11. Ethernet board connections and setup

The two boards have several indicator LEDs:

On the Ether1 board:

- D1:BSTATUS - Indicates a (PCMCIA) bus access by the ethernet chip - can be software configured to be driver-controlled.
- D2:LANLED - Indicates Network traffic (receive, transmit or collision);
- D3:LINKLED - UTP Link detected - can be software configured to be driver-controlled.

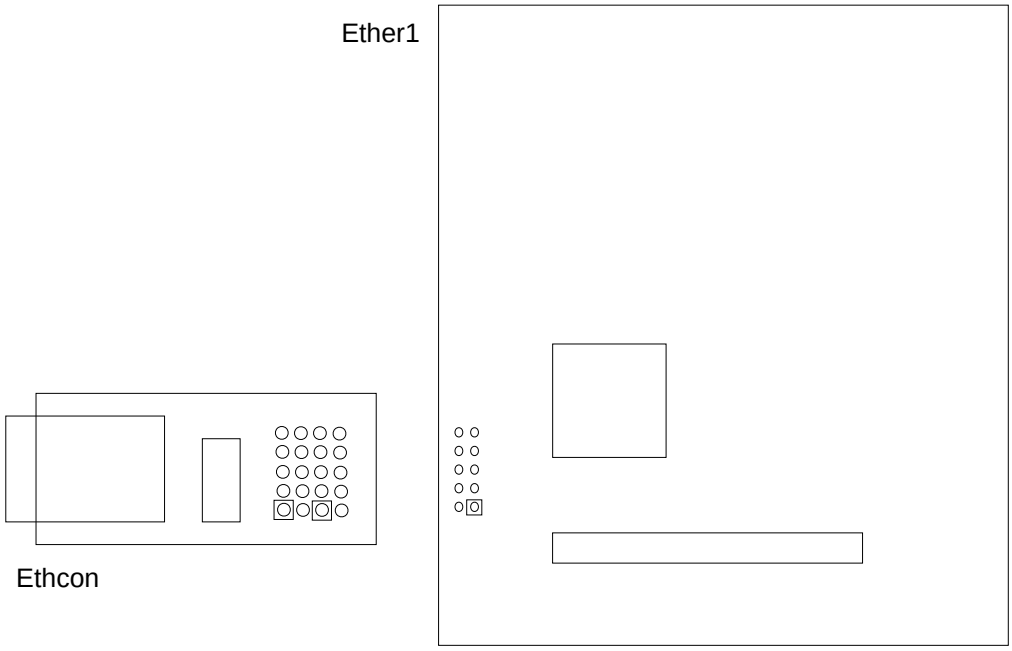
On the Ethcon board:

- (not labelled):LINKLED - UTP Link detected.

3.3.11.1. Ethernet config

The Ethcon board can be wired to the Ether1 board in a number of ways. Aleph One supply a suitable IDC cable, in which case arrange both boards with the connectors pointing towards each other and the solder side up, then connect the cable between them.

Various other physical arrangements are possible - the actual pinouts and some plausible physical arrangements are detailed below.



Layout of Ether1 and Ethcon showing pinouts - square indicates pin 1

The signals between the two boards are as follows:

Table 3-1. Ethernet board connections

Pin number	Signal
1	GND
2	RX-
3	RX+
4	LED
5	TX+
6	TX-
7	GND
8	GND
9	+5V
10	+5V

Pin layouts, both boards viewed from (main) component side:

Table 3-2. Ether1 board pinout

1	2
3	4
5	6
7	8
9	10

Table 3-3. Ethcon board pinout

row	1	2	3	4
	2	1	2	1
	4	3	4	3
	6	5	6	5
	8	7	8	7
	10	9	10	9

The double connector with reversed numbering means you can put a connector on the front in either rows 1 and 2 or 3 and 4 or on the back in rows 2 and 3 to get a match with the ether1 connector.

With an angle plug on the front of the ether board then a straight socket on the front of the ethcon interferes with the serial port connector, so the best place for the socket is the back of the ethcon (rows 2 and 3). This leaves the ethcon sticking up above the back of the LART.

An alternative would be to put a straight plug on the back of the ether1 board, and then a straight socket on the back of the ethercon board in rows 1 and 2, so the ethercon board lies flat.

3.3.12. Resources

A number of useful schematics for the mainboard, kitchen sink, ethernet board and JTAG programmer are available in PDF format on the CD and from the LART web site. On the CD they are all in LART/Hardware/ and only the schematics for the version of hardware in production are present. If you want to see the older schematics then you'll need to go to the web site.

Mainboard

LART Revision 4 mainboard schematic

CD: `/hardware/LART/Lart-rev-4.pdf`.

Web: `Lart-rev-4.pdf` (http://www.tudelft.nl/lartware/plint/Lart-rev-4.pdf).

LART Revision 3 mainboard schematic

`Lart-rev-3.pdf` (http://www.tudeflt.nl/lartware/plint/Lart-rev-3.pdf)

Kitchen Sink Board (KSB)

CPLD U9 Revision 2

`ksb_rev2.U9.pdf` (http://www.tudelft.nl/lartware/ksb/ksb_rev2_U9.pdf)

CLPD U1 Revision 2

`ksb_rev2_u1.pdf` (http://www.tudelft.nl/lartware/ksb/ksb_rev2_U1.pdf)

Kitchen Sink Board Revision 2

`ksb_rev2_u1.pdf` (http://www.tudelft.nl/lartware/ksb/ksb_rev2.pdf)

Ethernet Board

Ethernet Card, four channel, revision 1

`ether4_rev1.pdf` (http://www.tudelft.nl/lartware/ethernet/ether4_rev1.pdf)

Ethernet Card, single channel, revision 1

`ether1_rev1.pdf` (http://www.tudelft.nl/lartware/ethernet/ether1_rev1.pdf)

JTAG

JTAG dongle schematics

`jtag-lart_schematic.pdf`
(http://www.tudelft.nl/lartware/jtag/jtag-lart_schematic.pdf)

JTAG dongle Bill-of-materials

`jtag-lart_Rev_X1_BOM.txt`
(http://www.tudelft.nl/lartware/jtag/jtag-lart_Rev_X1_BOM.txt)

JTAG dongle hardware distribution with gerber files

`jtag-lart_Rev_X1.tar.gz`
(http://www.tudelft.nl/lartware/jtag/jtag-lart_Rev_X1.tar.gz)

Jflash

`jflash-linux.tar.gz`
(http://www.tudelft.nl/lartware/jtag/jflash-linux.tar.gz)

Chapter 4. Installing ARMLinux on the Assabet

This chapter describes how to get your Assabet running ARMLinux.

4.1. Assabet: Background

The Assabet is an SA-1110 evaluation board which may be repackaged as a PDA. Notable features include:

- The availability of programming utilities like the JTAG Programming Software V0.3 (for Windows) and Jflash-Linux (for Linux), and the boot loader Angelboot.
- Neponset - a SA-1111 Development module with many additional interfaces.
- Use of the 2.4 kernel series with support for the Assabet and Neponset via a patch chain.
- The capacity to run MicroWindows although a backlight power source needs to be added on ’build phase 4’ or earlier boards.

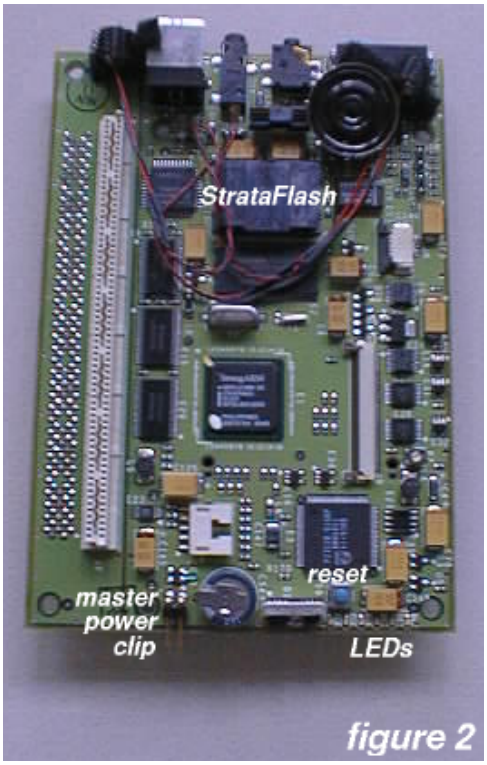
There is also support for Compact Flash on the Assabet and on PCMCIA support for the Neponset. In addition, the Socket LP-E CF + Card is now operative as is the IBM Microdrive.

4.1.1. Hardware Setup

> **Note:** The behaviour of the hardware, boot loader and kernel can vary because of the presence of Neponset - the Microprocessor Development Module. Readers who intend to use Neponset with Assabet should therefore refer to Section 4.1.1.4.

4.1.1.1. Preparing the Board

- Carefully remove the Li-ion battery and LCD/touchscreen by releasing the fasteners on the board receptacles.



Top view of Assabet board

4.1.1.2. Install Angel

On Assabet (and Neponset), the boot loader can be installed by physically inserting the pre-programmed flash chips supplied with the kit.

- Determine whether the board has a WindowsCE boot loader pre-installed by opening the StrataFlash sockets and examining the labels on each memory component. For precise details of the process of manipulating memories, refer to *The Intel StrongARM SA-1110 Microprocessor Development Board Users' Guide* section 2.3. (See Section 4.1.9 for the appropriate URL).
- Remove all power from the board and remove the jumper from the master power clip if it is installed (this disconnects the battery power). Open Socket E11 (StrataFlash) by sliding the middle panel away from the notch before lifting the doors. Socket E12 can also be opened in a similar way. (E12 is directly opposite to E11 on the other side of the board).
- The WindowsCE parts have a label which is similar to "WinCE E-Boot V3.0 L". Remove these parts with the suction part selector (similar to a pen supplied within a tube) and replace them with the parts which are labelled "Angel DB1110B 207D L".

4.1.1.3. Power and Serial Connections

- Power, JTAG, RS232 and POTS connections are made through base station - a 14-pin header. Connect the serial cable to your development workstation which - on a SA-1110 - means that the header should be attached to serial port 1.
- LED D4 should light up when the power is applied. Angel will then activate LED D8. After interrupt initialisation, Angel will strobe LED D9 four times. Supplementary information on the reset sequence can be obtained from `DB1110_diag.txt` which is bundled with Angel (http://developer.intel.com/design/strong/swsup/SA1110AngelKit.htm).

4.1.1.4. Install Neponset

- It is possible to attach the SA-1111 Development Module by using either of the 140-pin connectors which are found on the Assabet.
- Make a record of the *switchpack SW2 settings* because these affect selection, bus speed and other important parameters. A comprehensive explanation of switch meanings can be found in the `readme.txt` which is included in the Angel distribution although a summary of switch meanings is provided in the following table.

Table 4-1. Switch Options for Switchpack SW2

switch	off	on
SW2-1	reserved	reserved
SW2-2	reserved	reserved
SW2-3	debug port is UART3	debug port is UART1b (on Neponset)
SW2-4	bus speed is 103MHz	bus speed is 51MHz (clamped to 51MHz if PLL==147MHz)
SW2-5	flash device page mode enabled	flash device page mode disabled
SW2-6	Angel fast boot mode	Angel debug mode (see `DB1110_diag.txt`)
SW2-7	PLL==206MHz	PLL==147MHz
SW2-8	boot from flash on Neponset	boot from flash on Assabet

Note: All switches can be left in the *off* position except for the SW2-8 which can be left on.

4.1.1.5. Fixing Neponset

Some copies of Neponset may contain a bug involving the on-board Philips UDA1341 audio codec. The bug involves the L3 MODE signal coming from the SA-1111 which is available from PWM1 - the dual-function pin. Phase 4 Neponset boards incorrectly route this signal to PWMO which prevents any configuration of the codec from software. However, this problem may be solved by unsoldering the jumper wire at J7 and moving it immediately adjacent to J4.

4.1.2. Getting a Cross Compiler

We recommend that you install the emdebian cross-development environment supplied. See Section 8.2.2.

4.1.3. Getting a kernel

There are several pre-compiled kernels available on the CD and online. When you want to compile your own refer to Section 7.2. The correct default config setting for Assabet is: **make assabet_config**, and if you are using Assabet and Neponset together than it is **make neponset_config**.

4.1.4. Uploading to the Assabet

1. Power it up and connect to the serial port.
2. Get the Angelboot code. The precompiled version should work but you will need to be root. See Section 4.1.5.
3. Copy `dot.angelrc` and `Angelboot` from `Assabet/tools/angelboot` on the CD to your development directory and rename `dot.angelrc` to `.angelrc`. `.angelrc` is now classified as a *hidden file* so your filer may hide it from you. You will however, need to edit `.angelrc` so that it has the following settings:

```
base 0xc0008000
```

```
entry 0xc0008000
r0 0x00000000
r1 0x00000019
device /dev/ttyS1
options "9600 8N1"
baud 115200
otherfile ramdisk_img.gz
otherbase 0xc0800000
#exec minicom assabet
```

Note: You should change *device /dev/ttyS1* if your cable is plugged into ttyS0.

4.1.4.1. Configuring Minicom

In order to configure Minicom, take the following steps:

1. run it with **minicom -o** to get to setup;
2. set it to VT102, serial port ttyS1 (matching the `.angelrc` file) 9600baud 8N1 (8 data bits, no parity, 1 stop bit), no modem init string, no hardware handshaking;
3. save that as *assabet* config. Running Minicom as **minicom assabet** will run it up with these options.
4. If you don't use Minicom for anything else then you can make these settings the overall default - Minicom calls it `dfl` - settings.

OK. Now we are ready to go. Make sure the Assabet is powered, reset, that you have the 4 flashes on the orange LED and that the serial port is connected. Run:

./angelboot [image]

This will read the config from `.angelrc` before uploading the kernel and RAMdisk images. You should see the upload indicated, followed by several lines of stars during the kernel upload, then dots during the RAMdisk upload. See Section 6.8 for more details on how angelboot is configured.

If everything works OK you should see a penguin and a login prompt on the touchscreen which will disappear after a few minutes. Hitting **Return** in Minicom (start it if it hasn't already with **minicom assabet**) should get you to a root prompt from where you can login as **`root`**. You will not need a password.

daa-daa!

4.1.5. Re-flashing your Assabet

If your angelboot copy in strataflash becomes corrupt for any reason you can put angelboot back again by using the JTAG interface.

To use the JTAG interface, you will need:

1. to connect the custom 14-pin connector on the JTAG/serial/phone cable to the Assabet;
2. to connect the JTAG part (25-pin D connector) of the JTAG/serial/phone cable to your parallel port. Nothing else should be using the parallel port at the same time and you will need to be root as well;
3. to connect the power supply to the Assabet and power it up.
4. the Jflash-linux code (or Jflash - the Windows version). The pre-compiled binary supplied on the CD in `Assabet/tools/Jflash/`should work OK;
5. the binary of the boot loader which you will be putting in flash. Angel.bin is supplied with the Jflash distro.

To place the boot loader in flash, type:

./Jflash-linux Angel.bin

When this process has finished, reset the Assabet and the LEDs should show everything is working OK.

4.1.6. RAMdisks

A selection of useful RAMdisks is provided on the CD in `Assabet/RAMdisks`. These are the files that you upload using the 'otherfiles' entry in Angelboot.

See Section 7.3 for details of making your own.

4.1.7. Using the Audio Devices

Using the ramdisk_ks RAMdisk it is possible to play a sound by issuing the following command:

cp <file> /dev/dsp

where the file is a 44kHz stereo WAV file. Well, you can play any file you like this way, but it'll sound horrid if it's not a WAV file.

4.1.8. PCMCIA Support

For PCMCIA to work you need kernel PCMCIA support compiled in and the *cardmgr* daemon in the RAMdisk and running. This is included on ramdisk_ks (2.5MB - expands to 6.8MB) which is on the CD. Use this RAMdisk if you want to use the CF slot on the Assabet.

To get a cf disk recognised, simply plug it in. You can then check it with the **cat /proc/bus/pcmcia/00/stat** command.

The disk can then be mounted in the following manner:

mount /dev/hda1 /mnt

> **Note:** If you just pull disks out then things can get rather broken (and you may have to reset). You therefore need to **umount** the disk first.

4.1.8.1. The CF Network Card

You need to configure the card for your network environment in `/etc/pcmcia/network.opts`. You can then fill in IP and gateway details or simply use DHCP if it is available. Then, when you plug in the card, the eth0 interface is automatically created and the script is run to **ifconfig** the device. If you fill in an NFS mount in `etc/fstab` and specify the `mount` option in `/etc/pcmcia/network.opts`, the NFS mount is *automatically* done when you plug the card in and suddenly the network is available to you, making development and testing much easier.

4.1.8.2. Copying Things off a CF Disk

It is possible to copy things off the CF disk in a situation where they won't fit onto the RAMdisk.

You can use the rest of the RAM (if some is free) by doing:

mke2fs /dev/ram1 (creates an ext2 filing system in RAM)

mkdir /mnt/ram

mount /dev/ram1 /mnt/ram (mounts the RAMdisk)

This defaults to creating an 8MB RAMdisk. Now you have 8MB of 'disk' available to copy things into which will remain until you power down. You can just copy things out of `/dev/hda1/<path>/<file>` to `/mnt/ram/<file>` up to the size of the new RAMdisk.

4.1.9. Useful Resources

Additional information can be obtained from:

- *The Intel StrongARM SA-1110 Microprocessor Development Board Users' Guide* (http://developer.intel.com/design/strong/guides/278278.htm).
- *The Intel StrongARM SA-1110 Microprocessor Developer's Manual* (http://developer.intel.com/design/strong/manuals/278240.htm).
- The Institute for Complex Engineering (ICES) *Technical Report* (http://www.cs.cmu.edu/~wearable/software/docs/assabet-linux-report/).

Chapter 5. Installing ARMLinux on the Psion 5

This chapter describes how to install ARMLinux on your Psion 5 or Geofox One.

5.1. Introduction: Background

The Linux7k project is an effort to port the widely used Linux kernel to the Psion Series 5 and later pocket computers.

So far the team has ported the Linux 2.2.1 Kernel to the Series 5 and Geofox One and created Arlo and several initial RAMdisks (initrd) for the project. Work then stalled for a while as Psion were slow to release hardware information for later machines, but is now progressing again with the 2.4.1 kernel now booting on a Series 5mx.

5.1.1. Psion Series 5 Specifications

Specs

Memory

4 or 8MBs.

Processor

CL-PS7110 ARM Processor 18 MHz.

Display Resolution

640x240 (Half-VGA).

Display Type

Monochrome Touch-Screen (16 shades).

Removable Media Type

Compact Flash™ (CF) Disks.

Default OS

EPOC32 1.00 or 1.01.

5.2. How Does it Work?

- Boot loader - The boot loader, Arlo does for EPOC what Loadlin does for DOS. It releases the memory from EPOC, places the kernel and initrd in the appropriate places in memory, and then boots the Linux kernel.
- Kernel - The kernel has patches applied to it, which makes it compile for ARM based systems as well as having a few more patches for the Psion Hardware. The kernel port to the Series 5 is basically finished now.
- Initrd - This is a virtual filing system image, which contains a miniature version of Linux. These can range in size but the more RAM used for the RAMdisk will leave less for the operating system. The kernel loads these images as the root partition.

5.3. Preparation

You will need:

- a backup of your Psion’s disks (this is very important as *everything* in your C: drive will be lost; your D: drive should escape, but don’t count on it;
- the Psion installer;
- Arlo for your model of Psion (5 or 5mx);
- a glued kernel image;
- a compressed initrd;
- a paperclip or similar, thin, blunt item (to reboot your machine).

5.4. Booting using Arlo

5.4.1. Copying Files onto the Psion

The easiest way to get the files onto your Psion is to use the transfer software for your host machine. This is PsiWin for Windows, Plp tools for GNU/Linux and PsiFS for RISC OS. Use this to transfer the necessary files onto the Psion's RAMdisk or CF through EPOC. For example, to copy the `arlo.sis` file onto the Psion's C: drive (RAMdisk) using plptools you type the following command:

rfsv write arlo.sis c:/arlo.sis

5.4.2. Installing Arlo

You need to install Arlo (the boot loader - equivalent to LILO) onto your Psion. To do this, extract `arlo-0.51.tgz` and copy the `Arlo.sis` file to your Psion, then copy `INSTEXE.EXE` and run it. This should change the icon for `Arlo.sis` which you can then run and it will install itself to either C: or D: as you select - it will work fine from either. You need to reinstall it each time you boot Linux. If you install it to a CF disk, you won't have to re-install each time - Section 5.6

5.4.3. Getting the Kernel

You need a kernel for this project. I would recommend trying one of Werner's precompiled ones. A good one to try is `stable-221-cb23-519-psi.gz`.

Next, uncompress the glued kernel image with:

gunzip <glued image name>

and copy the `glued-*` and `initrd-*.gz` files to `C:\` on the Psion as `image` and `initrd.gz`.

5.4.4. Getting the Kernel from EPOC

5.4.4.1. Loading the Kernel

At this stage you are ready to boot into Linux. Double check you have good backups of your Psion and then double-click `Arlo.exe`. You will see:

```
Loading logical device
```

```
Opening the channel
Commands are:
...
Option buffer at address XXX
Memory description at address YYY
> _
```

Now type:

l image

and you should see:

```
Opening file image
Succeeded
File size XXX
Loading XXX bytes to address YYY
Read XXX bytes
No pages = ZZZ
> _
```

5.4.4.2. Loading the initial RAMdisk (initrd)

Now type:

l initrd.gz

and you should see:

```
Opening file initrd.gz
Succeeded
File size XXX
Loading XXX bytes to address YYY
Read XXX bytes
No pages = ZZZ
> _
```

5.4.4.3. Passing Additional Parameters into the Kernel

If you need to pass any other parameters into the kernel at this point (for example an alternative root filesystem) you need to type:

o root=/dev/hda2

A full description of kernel parameters can be found in `/usr/src/linux/Documentation/kernel-parameters.txt`.

5.4.4.4. Running the Kernel

Now type:

b

Your Psion should now boot into Linux!

If it does a double bleep and then puts up the EPOC logo, something has gone wrong. Try reinstalling Arlo (from the `Arlo.sis` and `INSTEXE.EXE` files).

5.5. What can I do Now?

Good question. Start by mounting `/proc`, it's satisfying and actually useful.

Type:

mount -t proc /proc /proc

You can then **ls /proc** if you want, good, eh?

You can also mount CF disks with:

mount -t msdos /dev/hda1 /mnt

You will get some errors when you first mount the disk and long filenames will produce errors when they are **ls**'d.

If you are feeling brave you could:

dmesg > /mnt/dmesg.txt

But you run a greater risk of trashing your CF disk.

5.6. Creating a Filesystem on your Compact Flash

If you want to boot your system off a CF disk, you need to repartition it. You will need a reasonably sized CF though (over 16MB).

First, you boot into `initrds3.gz` with a kernel that supports CF disks. When this has booted up, you need to execute **fdisk /dev/hda**. You can then repartition your system. Ideally you should make a 4-6MB Fat16 partition as /dev/hda1, and then assigning the rest for Linux (/dev/hda2). When you have done this, boot into EPOC and format the CF. Install Arlo, a kernel and an initrd in the root of this partition. Now you don't have to re-install Arlo every time you reboot.

To boot a CF based system, you need to pass the appropriate option into the kernel (see Section 5.4.4.3), alternatively, you will need to use the **rdev** command to alter the flags in your homebrew kernel before you glue it.

5.7. Autobooting Arlo from Reset

To automatically boot Arlo (or in fact any EPOC program) you must create a `\system\data\wsini.ini` file on your CF FAT partition; it should contain the following line:

STARTUP d:\arlo.exe

This presumes you have the Arlo executable on the D: drive (compact flash) and the associated Arlo.

5.8. How do I get Back to EPOC?

Type:

sync
sync
sync

Close the machine.

On the Psion:

Open the battery backup door and locate the small copper coloured circle near the battery, using a partly unfolded paperclip or similar, gently press in the copper coloured circle. Now close the backup battery door, and, while holding down both shift keys press the **Esc**/**on** key. The machine should bleep once and then display the Psion 5 splash screen, after a delay you will be back in the EPOC system.

On the Geofox: (info supplied by Hennie Strydom).

It is mostly identical to the Series 5, with the single exception that the reset switch is next to the speaker on the bottom of the Geofox.

5.9. Rolling your Own (Cross Compiling)

5.9.1. Compiling a Kernel

Alternatively, you can create your own kernel, using the cross compilation environment described earlier. To compile a custom kernel, you will need to download the following files:

- `linux-2_2_1.tar.gz`
- `patch-2_2_1-rmk2.gz`
- `linux-2_2_2-philb-990208.gz`
- `crash+burn-26_patch.gz`

Assuming that you have downloaded these and copied them to `$HOME/Linux7k`, execute the following commands:

cd $HOME/Linux7k
gunzip linux-2_2_1.tar.gz
tar -xvf linux-2_2_1.tar
zcat patch-2_2_1-rmk2.gz | patch -p0
zcat linux-2_2_1-philb.gz | patch -p0
cd linux
zcat ../crash+burn-26_patch.gz | patch -p1

These commands have now patched the Linux kernel for use with the Psion Series 5.

You must now configure and compile the kernel. To configure it, execute:

make xconfig

Or, if you don't have X Windows, execute:

make menuconfig

When you have done this, execute this command to make the kernel:

make Image

This command will create a kernel image in `arch/arm/boot/Image`.

5.9.2. Glueing the Kernel

You must now *glue* the kernel, so that it can boot on the Psion. This basically adds the ARM *bootstrap* code onto the front of the kernel. To do this, execute the following commands:

cd ../

gunzip boot-13.tar.gz

tar -xvf boot-13.tar.gz

cd boot

make ./glue.pl ../linux/arch/arm/boot/Image > ../kernel_image

These commands will now make a glued kernel image in `$HOME/Linux7k/kernel_image`. You may now use this kernel image as your kernel.

5.9.3. Debugging

5.9.3.1. GDB Stub

There is a GDB stub available from the sourceforge ftp site written by Noel Cragg. The `README` details all you need to know.

5.9.3.2. Armulator

To simulate the Psion 5 on your desktop with full debugging capability you can use the http://staticip.cx/~benw/linux7k by Ben Williamson, cool bit of kit.

5.10. Resources

5.10.1. Web sites and Mailing Lists

The following web site is the place for Psion Linux port info project:

- http://linux-7110.sourceforge.net/

5.10.2. Mailing Lists

The project has a mailing list <`linux-7110@sourceforge.net`> which is the primarly mechanism of communication and information on how the project is

progressing. To subscribe, go to
http://lists.sourceforge.net/lists/listinfo/linux-7110-psion.

5.10.3. Plp Tools

This is the easiest program to transfer files from your Linux desktop to the Psion, pretty much plug-n-play.

- ftp://linux-7110.sourceforge.net/pub/linux-7110/Plp
- http://www.proudman51.freeserve.co.uk/psion/index.html

5.10.4. EPOC Program Installer (Instexe.exe)

Included in the Arlo-0.51 distribution.

5.10.5. Arlo

5.10.5.1. For the Psion 5

- ftp://linux-7110.sourceforge.net/pub/linux-7110/Arlo/Arlo-0.51.tgz

5.10.5.2. For the Psion 5mx

- ftp://linux-7110.sourceforge.net/pub/linux-7110/Arlo/Arlo-1.0.tgz

5.10.6. Precompiled Kernels

- ftp://linux-7110/sourceforge.net/pub/linux-7110/KernelBinaries
- ftp://lrcftp.epfl.ch/pub/people/almesber/psion

5.10.7. Initial RAMdisks (initrd’s)

- ftp://linux-7110/sourceforge.net/pub/linux-7110/Klaasjan/initrds
- http://crosswinds.net/~linux7kj
- http://linuxhacker.org/pub/flatcap
- ftp://lrcftp.epfl.ch/pub/people/almesber/psion

Chapter 6. Tools and Utilities for Booting

6.1. Boot loaders

Booting on any device needs a boot loader and some means to get the boot loader installed in the first place. Here we look at the specifics for ARM architecture devices booting Linux, covering available boot loaders and their capabilities, some general principles and mechanisms for uploading the Linux kernel and root filesystem, giving specific examples with Jflash-linux, Angelboot, and Blob.

The first software to be present on a virgin piece of hardware are the kernel and boot loader. They are often installed from an external device such as a PC or Linux machine, via the serial or parallel port. Once present on the target hardware, other boot mechanisms become available.

6.2. Introduction

In this chapter we are looking at the stuff that happens *before* the Linux kernel boots - loading and executing the kernel with a boot loader, and getting a boot loader onto the device in the first place. I will also cover some related aspects as we go along such as wake-up from sleep, which needs close co-operation between the kernel and boot loader.

I will endeavour to cover things in enough detail that you understand the underlying principles and could happily go away and boot ARMLinux on a supported device in a few hours, and get it booting on an unsupported (by the boot loader) device in a few days.

The ARMLinux kernel now supports a huge range of ARM devices; 114 as of September 2001, and the rate at which new devices are being added is accelerating. Along with the host of different ARM devices comes a host of boot loaders - I’ll mention all the ones I’m aware of but only cover a couple in any detail.

6.3. Boot loader Basics

There are many variables to consider when talking about boot loaders:

- the medium they are installed in (usually ROM, EEPROM or Flash);
- the boot loader software used (e.g. Blob, bootldr, Redboot, Angel);
- the device/mechanism used to control the loader (none, serial port, switches);
- the storage device from which the kernel image is downloaded (host PC, disk, CF, internet);
- the device and protocol by which the image is transmitted (serial, USB, ethernet, IDE);
- the 'host-side' software used to transmit the image, when it is coming from an external device/machine;
- whether the boot loader is single-stage or multi-stage. Booting from disk is a 2 or 3 stage process, and some devices have a native boot loader/OS that can't easily be replaced so that must be used to fetch and boot the Linux kernel;
- the CPU type;
- 'sleep' resets.

All of this makes it difficult to generalise. However in practice there are a number of common scenarios which cover most cases. The commonalities found in many situations and devices mean that of the many boot loaders available there are now several which can run on a range of platforms. Each boot loader supports a different subset of download mechanisms and devices. This area changes quickly as most are still under active development but I will cover the current state of play and the pros and cons of different boot loaders so you can see which one is most likely to suit your purposes.

To boot Linux, the boot loader has to load two things into memory; the Linux kernel ('kernel') and a root filesystem ('root') for it to use. It has to do a few other things too, but we'll come to those later. It is possible to link the kernel and boot loader together into one file, which makes the bootloading part simpler and the linking significantly more complicated. It's only worth considering if you have a loader that can only load one file and no easy way to change it. The filesystem usually takes the form of an initial RAMdisk (or 'initrd') which is a gzip-compress normal ext2 filesystem image, but it can also be a RAMFS, CRAMFS or JFFS2 image. The pros and cons of these various types are covered elsewhere and I will generally assume that you are loading a RAMdisk image here.

Most boot loaders are capable of two distinct modes of operation - 'Bootloading' and 'Downloading'. In principle there is no big difference between these modes, but

in practice it is useful to consider them separately as they feel quite different to the engineer.

Bootloading

- In this mode the target device operates autonomously, loading its kernel and root filesystem from an internal device - i.e. the normal booting procedure.

Downloading

- In this mode the target device downloads the kernel and root filesystem from an external device, often under the control of that device. This is the mode that is used to first install the kernel and root, and for subsequent updates.

Where the same boot loader is responsible for both these tasks there has to be some way to change the behaviour between these two modes. This maybe a link on the board, or the reception of a character on a serial link which interrupts the normal boot and puts the boot loader into a mode where it will take commands (this is how Blob operates).

6.4. Boot loader Responsibilities

So, before getting further into the nitty-gritty, what does the boot loader need to do before handing over control to the Linux kernel? I am assuming here that the kernel and root will be running from RAM. It is possible to configure things so that much of the kernel remains in ROM or flash (at the expense of a great deal of speed). The root filing system can also be already set up to run from flash and thus not need loading explicitly. In the latter case this is simply a matter of not doing the things below associated with loading a filesystem - the kernel just needs to know what sort it is and where to look; either as command-line options or compiled-in defaults.

The bootloader needs to:

1. initialise base hardware - CPU speed, memory timings, interupts, detect RAM size;
2. initialise any devices it needs to use for reading the kernel and root images;
3. arrange a block of contiguous physical memory for the kernel, starting at a particular address, and another for the root, also contiguous, but not necessarily adjacent to that for the kernel;
4. copy the kernel and root to their contiguous areas;
5. call the kernel with r0=0, r1=machine architecture number with the MMU and caches turned off.

From there on we are in the ARMLinux startup procedure.

In practice there are a couple of other things that the boot loader is likely to do. It will normally initialise things like a serial UART and at least one GPIO for communication purposes. The GPIO is typically used to drive an LED (there maybe several) to give out simple OK/Error codes. The serial line is usually used to send out status and receive boot loader commands. These things will be re-initialised later by the kernel if the boot loader doesn't do it so if it doesn't need them it can leave them alone. It should also check that the kernel image is valid before trying to execute it.

The requirement for the boot loader to detect memory is only for devices where the amount of memory may vary. Where it is fixed it can be hard-coded into the kernel, but otherwise it should be passed as a kernel parameter.

For the boot loader to be able to pass parameters to the kernel they need to agree on a memory location for this information to go. Prior to kernel 2.4 a structure was used for information to be passed to the boot loader, but this was unwieldy and prone to break when new options were needed. So for 2.4 kernels onwards a tagged-parameter mechanism is used, which means that new tags can easily be added without needing a close matching between boot loader and kernel versions. The boot loader and kernel do still have to agree on the memory location at which the parameters will be stored. For the SA11x0 platforms the de facto standard is 0xc0000100 (i.e. 256 from the bottom of physical RAM).

The 'arranging contiguous memory' part is also effectively optional. For most devices once the memory controller has been set up with the RAM timings then all of physical memory is ready and available to have data copied into it in the right place. Because the MMU is still turned off things are simple. The problem comes for boot loaders running on devices that already have a native OS, such as the Risc PC (RISC OS) or Psion5 (EPOC). In these cases the boot loaders are native OS applications which have to rearrange the OS memory tables to make a space at the right physical address to put the kernel and root. They have the advantage that loading the kernel and root images is easy, as there is a handy OS all set up giving them functions to do it, but that same OS wants to hide the workings of memory page tables from its applications so this part can be quite involved.

Before calling the kernel the bootloader should check that there is a valid kernel. For a compressed kernel (zImage) the things to check are in Table 6-1. If the magic number is missing then the kernel is missing, corrupt or not compressed. The link address should be the same as the place to which the zImage was loaded and will be executed from. The end address allows the loaded to check the file it loaded was long enough.

Table 6-1. zImage (compressed kernel) identifiers

Offset	Value	Meaning
0x24	0x016f2818	magic number identifying an ARM zImage binary
0x28	0xa0200000	link (or load) address for this zImage
0x2c	0xa02b4ccc	end address for zImage

> **Note:** The requirement for the caches to be turned off when the kernel is called is slightly relaxed on the StrongARM and Xscale, where only the data-cache needs to be off. The separate instruction cache can be On or Off.

Another little wrinkle is that the kernel, which is normally loaded at the bottom of physical RAM, needs to be loaded with a 32K (0x8000) space below it. This is used by the kernel page tables. (The actual requirement is that the kernel is in the 256MB chunk at the bottom of RAM, which is pretty easy to achieve!). So, if your RAM is at 0xc0000000 then the kernel load address is normally 0xc0008000. The root image is conventionally loaded 8MB above this at 0xc0800000. The considerations when deciding how to use memory during the boot are:

- the kernel and RAMdisk images should not overlap with the decompressed kernel;
- after the kernel is decompressed, the compressed image is discarded, so that part of memory is free for other use;
- the total size of decompressed kernel + RAMdisk image + decompressed RAMdisk should fit into memory;
- after the RAMdisk is decompressed the kernel discards the compressed image usually freeing a couple of megabytes;
- the kernel image, RAMdisk image, and decompressed kernel must not overwrite the boot loader parameter list.

Most boot loaders actually have significantly more functionality than this and are capable of accepting commands and acting on them, usually over a serial interface. We'll look at an example below. Fancier boot loaders can also take kernel boot options and pass them on to the kernel (as the popular PC GNU/Linux boot loader LILO does).

And finally the boot loader has to consider sleep modes. The ARM executes the instructions at physical memory location 0 on a reset, and this is where boot loaders live. However it also does this when it is awakened from sleep (or if the watchdog

causes a reset). For the sleep functionality to work the boot loader has to know to check the appropriate status register, and not to do a normal reboot if this was a sleep wakeup. Instead it should follow the wakeup-from-sleep procedure. On the SA11x0 this means checking the Reset Controller Status Register (RCSR) to see what sort of reset it was and if it was wake-up from sleep then get the value from the Power Manager Scratchpad Register (PSPR) and jump to it (under ARMLinux this register is used to hold the address of the wakeup code). The detail of this will vary for different implementations of the ARM core by different manufacturers.

6.5. Booting the boot loader

Of course, before the boot loader can do its thing you need to get it installed on the target hardware. That can be done with an old-fashioned ROM or EEPROM but on modern ARM platforms it is done using a JTAG uploader. JTAG was conceived as a mechanism for testing chips but has developed into a full blown debug system. The facility which lets it be used to test all the pins means it can also be used to execute instructions by feeding the right signals into the JTAG port. Thus, given a JTAG port you can start executing instructions on a CPU as a bootstrapping mechanism, and use those instructions to download a boot loader and store it in flash or EEPROM.

Platforms with a native OS don’t have this problem - the boot loader application is installed like any other application.

The uploaders I am aware of are Jflash-linux (or Jflash, the Windows version), used with StrongARM 11x0 processors, and Shoehorn, designed for the ARM7xxx series. Jflash uses the JTAG interface. Shoehorn relies on the built-in boot-ROM on the board, and actually downloads over a serial or ethernet interface.

> **Note:** As the JTAG mechanism exposes the hardware at a very low level you will normally need a version of the Jflash code that is specific to the board - any significant change in CPU, memory map or other devices in the JTAG chain will need changes in the code.

The JTAG interface is not directly compatible with the PC parallel port so a small adapter or ’dongle’ is usually required to provide a bit of logic and protect the JTAG port from over-voltage from the PC. Intel supply a suitable cable with their boards and the LART project has a suitable design.

Here is an example of using Jflash-linux to put the Blob boot loader onto a LART board.

6.5.1. Using JTAG and Jflash-linux

In order to use the JTAG programmer, you need to connect the parallel port of the host to the JTAG port of the target device you want to program. Ensure the device is powered up. Jflash and Jflash-linux both auto-sense for available parallel ports so it shouldn't matter where you plug it in.

You can then issue the following command as root; then, after uploading, the device will reset itself before running Blob.

./Jflash-linux precompiled-blob-2.04

and you should see something like the following after which the LART will reset itself before running Blob.

```
using printer port at 378
Seems to be a pair of 28F160F3, bottom boot. Good.

Starting erase for    c8e bytes
Erasing block   0
Erasing done
Starting programming
Writing flash at hex address      1b0, 13.44% done
Writing flash at hex address      3d0, 30.37% done
Writing flash at hex address      5f0, 47.29% done
Writing flash at hex address      810, 64.22% done
Writing flash at hex address      a30, 81.14% done
Writing flash at hex address      c50, 98.07% done
Programming done
Starting verify
Verifying flash at hex address      643, 49.88% done
Verification successful!
```

Note: If something is wrong you will see error messages like the following:

```
error, failed to read device ID
ACT: 0000 0000000000000000 00000000000 0
EXP: X001 0001000010000100 00000110101 1

failed to read device ID for the SA-1100
```

or

```
error, failed to read device ID
ACT: 1111 1111111111111111 11111111111 1
EXP: X001 0001000010000100 00000110101 1
```

```
failed to read device ID for the SA-1100
```

or

```
error, failed to read device ID
ACT: 1001 1010000111001011 01011000111 1
EXP: X001 0001000010000100 00000110101 1

failed to read device ID for the SA-1100
```

There are a number of possible causes. If you get all 1’s then you maybe using the wrong parallel port, or the cable or dongle is not plugged in, or the device is not powered up. If you get all zeros then you maybe using the wrong version of Jflash-linux, or (on a LART) you maybe suffering from the ’JTAG reset problem’, which is fixed by adding an extra pull-up resistor to the dongle design. If you get random numbers some of the time and zeros the rest of the time then you are almost certainly using the wrong version of Jflash-linux - i.e. one intended for a different device, such as Assabet.

6.6. Summary of boot loaders

The section after this describes the particular examples of Blob and Angelboot.

It is difficult to keep up with the current state of play in this area as new devices are appearing so fast, but here is a list of all the software I am aware of, with a summary of its functionality, licensing, provenance, and supported hardware. Some notes about pros and cons are added where relevant.

ABLE. Boot loader, GPL. Written by Ben Dooks and Vince Sanders. Supports the RiscStation and Simtec A7500FE boards which are ARM7500FE based devices. Stored in ROM/EEPROM - loads images from a raw partition on IDE disk. Controlled via serial console when downloading. Please refer to http://kyllikki.fluff.org/software/able/.

Angel/AngelBoot. Target+host boot loader. Supports Intel Assabet. Target part stored in flash. Loads images over serial interface. Saves to flash. Host part configured with simple rc file. During normal booting it loads the kernel and RAMdisk from RAM before handing over to the kernel.

Angel and Angelboot together make a simple but effective boot loader that can upload a kernel and RAMdisk to a target device from a host, then run them. Angel is the code that lives on the device that gets started on boot and Angelboot is the program that runs on the host and manages the upload. Angel is in fact part of the

ARM SDT and can be used as a debugger in that environment. Angelboot allows it to be used outside this environment. Angel is 49K, Angelboot 34K

Blob. The Boot Loader OBject (Blob) has a slightly-modified GPL license, which makes clear that the loaded OS is not deemed to be a 'derived work'. It lives in local (or add-on) flash and is capable of uploading and saving a kernel, RAMdisk and itself over a serial port to flash and passing a kernel command line to the kernel. It has a command line interface which runs over the same serial channel - uploads are done as UUcoded files. During normal booting it initialises the hardware and loads the kernel and RAMdisk from flash to RAM before handing over to the kernel. Command mode is entered by sending a character down the serial interface during the 10 second 'autobooting....' period.

Blob was originally written for the LART by Erik Mouw and Jan-Derk Bakker, but has since had support added for a number of other platforms by others - the Assabet (Intel SA1110 evaluation kit), Brutus (SA-1100 evaluation platform), PLEB (SA1100), Shannon (TuxScreen), NESA and CreditLart (SA1110 successor to LART). To use it with Brutus or Assabet you should stick to v1.0.8-pre2 or earlier, as the major rewrite for the 2.x series (at least up to 2.0.4) has broken support for these - check the RELEASE-NOTES file for current status. You need to compile the correct version for the hardware you want to use it with as the hardware varies in important ways (processor, memory type and layout, debug serial port). Adding support for any new SA11x0 based device is very straightforward. Size 16K.

The Blob pre-compiled binaries are at:- http://www.lart.tudelft.nl/lartware/blob.The source and project mailing lists are at http://blob.sourceforge.net/, and it is currently being actively developed.

Bootldr. Very full-featured boot loader - probably the most comprehensive here. Used primarily on the Compaq iPAQ, but also on the Intel Assabet, Skiff and HP journada720 and for some other projects (e.g. Balloon). Boot parameters for both the loader and kernel stored in flash along with the loader, kernel and root filesystem (normally JFFS2, but RAMdisks and CRAMFS are also supported). Has about 30 commands to partition flash and control its use. Can drive an iPAQ framebuffer to display a splash screen, and its actions can be controlled by GPIOs (the buttons on a iPAQ). Partitions are defined in flash for the boot loader parameters, bootldr itself, kernel, filesystem image. Other partitions can be defined and cleared or loaded to.

Uploads are done using XMODEM over the serial port. Other functionality includes commands to read and write arbitrary RAM locations, and write arbitrary flash locations (with an override control so some areas are normally protected). Also options to set config for booting with an NFS-mounted root. It can also boot QNX, BSD and WinCE, as well as Linux. Size - 100K

The Compaq license is intended to allow both non-GPL and GPL variants to be compiled. The former is for use where the user wishes to include some proprietary code in the loader. Download the current stable version from here:

ftp://ftp.handhelds.org/pub/bootldr. Or go here for development code and mailing list: http://www.handhelds.org/source.html

Jflash. This is actually two related programs. The original Jflash which is a Windows program from Intel , and Jflash-linux, the Linux port by Nico Pitre. They are functionally identical in that they allow the downloading of a boot loader to a device through the JTAG interface. IT has an X11/BSD-type licence. Versions are available for Intel Assabet, LART, Shannon (TuxScreen). Note that each version is different as they are closely tuned to the hardware. The download is done via the host PC parallel port using a suitable dongle or cable.

The Windows Jflash code, or 'SA-1110 Development Board, JTAG Programming Software v0.3' to give it its full title, can be downloaded from: http://developer.intel.com/design/strong/swsup/SA1110Jflash.htm whilst Jflash-linux is obtainable from: ftp://ftp.netwinder.org/users/n/nico/SA1110Jflash.tgz. Size approx 34K

Hermit. Host+target boot loader, GPL. Ben Williamson/Mike Toumlatzis. Cirrus logic EP7110 and 7210 boards, EDB79812 with patches. Target portion is downloaded with Shoehorn. Then combines with host portion to save itself to flash and manage further downloads to flash over ethernet (raw packets) or serial.

A fairly straightforward boot loader that just does its job. Ethernet download makes it very fast in comparison to serial downloads. The ethernet support uses raw sockets (i.e. no TCPIP stack) and doesn't currently deal with collisions so it is best to have either a crossover lead between host and target or a hub with no other devices using it. It is normally used in conjunction with Shoehorn to get it initially downloaded. Size 32K on host, approx 20K on target. Refer to http://www.bluemug.com/~miket/arm/arm.html for further information.

NeTTrom. Boot loader, GPLed. Supports Corel/Rebel NetWinder. This loader is actually a small Linux kernel with hard disc and Network support. Lives in ROM, and downloads from IDE drive or network using DHCP and TFTP. Parameters are stored in flash. Autoboot is interrupted by a keypress on the console which will be a keyboard if present, or a serial port if no keyboard is detected. It provides numerous config options to specify the kernel file, root file, HD partition or NFS/network options. It can also save a new boot loader, or RAMdisk to flash, or files to disk/network. Due to being based on the Linux kernel the image is larger than most at approx 500K.

Redboot. Based on the eCos Hardware Abstraction Layer (it is in fact an eCOS application) and uses the eCos license. You can boot from flash or over a network, and can download images via serial (XMODEM, YMODEM) or ethernet (BOOTP/DHCP support for IP config, then TFTP for images). It has command line/scripting interfaces. Includes a GDB stub to allow communication with gcc/g++ apps running on the target. It supports booting Linux and eCos apps, and can be obtained from http://sources.redhat.com/redboot/.

Redboot has very wide device support including ARM7, StrongARM and Xscale devices, due to its eCOS heritage. As of September 2001 these devices were supported:

Embedded Performance Inc: EPI Dev7 (ARM7TDMI) and EPI Dev9 (ARM940T)
Cirrus Logic:EDB7211 (EP7211), EDB7212 (EP7212)
ARM: Evaluator-7T (KS32C50100),Integrator (ARM7TDMI)
Atmel: AT91EB40 Evaluation kit (AT91x40)
Intel:EBSA-285 (SA110), Intel SA-1100 Multimedia board (SA1100), Assabet (SA1110)
IQ80310 processor evaluation kit (Xscale)
Compaq iPAQ (SA1110)
Brightstar engineering: CommEngine (SA1110)

Shoehorn:. Downloader, GPL. Ben Williamson/Mike Toumlatzis. Cirrus logic EP7110 and 7210 boards, EDB79812 with patches. Works with built-in boot ROM to download to RAM via UART1 or ethernet. Normally used to download Hermit. Size 17K. Please refer to http://www.bluemug.com/~miket/arm/arm.html for further information.

6.7. Some practical examples

6.7.1. An Example Using Blob

Here is an example of how to use Blob to download the kernel and root filesystem (RAMdisk). We are working from the point where Blob has already been installed using Jflash-linux, and you have some kind of terminal device to talk to the serial port.

1. Connect a terminal (or a terminal emulator like miniterm or Seyon) to the target device's serial port;
2. use the appropriate settings for your device: usually 9600 8N1 or 115200 8N1; on the LART there is no hardware flow control so you'll need to turn that off too;
3. use the appropriate terminal emulation, usually VT100;
4. switch the target device on.

You should then see something like (this example is from a LART):

```
Consider yourself LARTed!
```

```
  blob version 2.0.4,
  Copyright (C) 1999 2000 2001 Jan-Derk Bakker and Erik Mouw
  Copyright (C) 2000 Johan Pouwelse
  blob comes with ABSOLUTELY NO WARRANTY; read the GNU GPL for details.
  This is free software, and you are welcome to redistribute it
  under certain conditions; read the GNU GPL for details.
  Memory Map:
    0x08000000 @ 0xC0000000 (8MB)
    0x08000000 @ 0xC1000000 (8MB)
    0x08000000 @ 0xC8000000 (8MB)
    0x08000000 @ 0xC9000000 (8MB)
  Loading blob from flash . done
  Loading kernel from flash ....... done
  Loading ramdisk from flash ............... done
```

Note: It always says `Loading kernel/ramdisk` whether or not a kernel and RAMdisk are actually present. It will just hang at `Starting kernel ...`if it tries to boot a 'blank' kernel.

```
Autoboot in progress, press Enter to stop ...
```

If you didn't press **Enter** within 10 seconds, Blob will automatically start the Linux kernel:

```
Starting kernel ...

Uncompressing Linux...done.
Now booting the kernel
...
```

However, if you press **Enter**, you will see the Blob prompt:

```
Autoboot aborted
Type "help" to get a list of commands
blob>
```

Blob comes with a selection of useful commands. Type **help** for a list or refer to Table 6-2.

Table 6-2. Blob Commands

Command with Options	Explanation
boot `[kernel options]`	Boot Linux with kernel options

Command with Options	Explanation
clock `PPCR MDCNFG MDCAS0 MDCAS1 MDCAS2`	Experimental speed setting option - USE WITH CARE!
download `{blob|kernel|ramdisk}`	Download blob or kernel or RAMdisk image to RAM
flash `{blob|kernel|ramdisk}`	Write blob kernel or RAMdisk from RAM to flash
help	Get this help
reblob	Restart blob from RAM
reboot	Reboot system
reload `{blob|kernel|ramdisk}`	Reload kernel or RAMdisk from flash to RAM
reset	Reset terminal
speed	Set download speed
status	Display current status

6.7.2. Compiling Blob

> **Note:** Two versions of pre-compiled Blobs are provided on the CD and the LART web site so most people will not need to compile Blob until you want to modify it or perhaps get a newer version.

Should you need to, here are the details. First you need a suitable GNU toolchain:

- A native ARM/StrongARM Linux system with gcc 2.95.2, and binutils 2.9.5.0.22 or better;
- or any Unix system with cross compiling binutils 2.9.5.0.22 (or better) and gcc 2.95.2 installed;
- GNU make (although some vendor supplied make utilities will do);
- GNU autoconf and automake (if you build Blob from CVS);
- a Linux kernel source tree: the latest linux-2.4.* kernel will usually do. If not, apply the appropriate -rmk patch. Linux-2.2.* kernels will NOT work, but these are considered obsolete for arm-linux anyway.

Next you need to get the software, then configure and compile it. The lastest version available at the time of writing is `blob 2.0.4.tar.gz` which is on the CD in `/tools/blob/`

This example assumes you will unpack it in `/usr/src`

cd /usr/src
tar -xzvf /cdrom/tools/blob/blob 2.0.4.tar.gz
cd blob-2.0.4

Now you need to specify the board for which you want to compile it, and whether you are cross-compiling or not, i.e. if you are compiling on a PC rather than a native ARM machine. There are currently four valid board names so choose from: 'assabet', 'brutus', 'lart', or 'pleb'. If the board name is omitted, lart will be chosen as a default. If you are modifying Blob itself then you should turn on the `--enable-maintainer-mode flag`. This will automatically regenerate Makefiles and configure scripts if you change the source files. You need autoconf (>= 2.13) and automake (>= 1.4) installed for this feature.

export CC=/path/to/cross/gcc
export OBJCOPY=/path/to/cross/objcopy
./configure --with-linux-prefix=/path/to/armlinux/source --with-board=boardname

So on our example host machine the correct runes are as follows :

export CC=arm-linux-gcc
export OBJCOPY=arm-linux-objcopy
./configure --with-linux-prefix=/usr/src/arm-linux/linux/ --with-board=lart

If things have worked you should get confirmation text like this:

```
Target board                lart
C compiler                  arm-linux-gcc
C falgs                     -O2 -I/usr/src/arm-linux/linux/include -Wall
Linker Flags                -static -nostdlib
Objcopy tool                arm-linux-objcopy
Objcopy flags               -O binary -R .note -R .comment -S
run-time debug information  no
```

For a native ARM machine it's much simpler - all you should need is this:

./configure --with-board=lart

So, now you just make Blob with:

make

This will result in a stripped binary `blob` in the `blob-2.0.4/src/` directory. There will also be some other executables - `blob-rest`, `blobrest-elf32`,

`blob-start` and `blob-start-elf32`. These are the two parts of both in full (`-elf32`) and stripped form. Blob is compiled in two parts that are linked together in order for it to be able to copy itself into RAM and restart, which it needs to do in order to write to the flash RAM. The much smaller `blob` is the one you actually upload. The `-elf32` versions are useful for examining the code produced with a disassembler. The best way to upload Blob to your device is using the appropriate version of JFlash-linux.

6.7.3. Resources

In addition to the Blob-related resources which are available on our CD, you can find:

- the Blob source and pre-compiled binaries http://www.lart.tudelft.nl/lartware/blob and on the CD.
- the latest Blob CVS version and developers mailing list at http://blob.armlinux.org/

6.8. Angel and Angelboot

Angel and Angelboot are an easy boot loader team to use, that illustrates the basics nicely. Here is an example of its use, in a host machine that has a terminal such as `minicom` or an emulator such as `seyon`, in uploading a kernel and RAMdisk to an Intel Assabet board and then running them.

Angel is the code that lives on the device that gets started on boot and Angelboot is the program that runs on the host and manages the upload. Angel is in fact part of the ARM SDT and can be used as a debugger in that environment. Angelboot allows it to be used outside this environment.

6.8.1. Configuring Angelboot

Copy `dot.angelrc` and `Angelboot` from the Angelboot distribution to your development directory and rename `dot.angelrc` to `.angelrc`.

`.angelrc` is now classified as a *hidden file* so your filer may hide it from you. You will, however, need to edit `.angelrc` to match your device, so that it has the following settings:

```
base 0xc0008000
```

```
entry 0xc0008000
r0 0x00000000
r1 0x00000019
device /dev/ttyS1
options "9600 8N1"
baud 115200
otherfile ramdisk_img.gz
otherbase 0xc0800000
exec minicom
```

The meaning of these settings is as follows:

- The base address is the address in target memory that the kernel is loaded to. The file loaded is `zImage` in the current directory;
- entry is the address in target memory that gets jumped to when the loading is complete. For the kernel this is normally the same as the load address;
- r0 is what will be in register r0 when the kernel is started - this is normally zero;
- r1 is what will be in register r1 when the kernel is started - this is normally the architecture number for the platform;
- device is the device to use for the upload - normally a serial port such as /dev/ttyS0;
- options are the serial port settings to use after the upload is complete - these should match your terminal settings;
- baud is the serial speed to use for the upload;
- otherfile is the name of a second file to upload - normally the RAMdisk;
- otherbase is the address in target memory that otherfile is loaded to;
- exec allows a command to be executed once the upload is completed - normally starting your serial communication program.

6.8.2. Running Angelboot

Make sure your device is ready and that the serial port is connected. Run:

./angelboot [image]

Angelboot will run using the config in `.angelrc`. You should see the following:

```
Initialising execution environment.
Downloading image size 0x98c34 to 0xc00080000
```

followed by 4 lines of stars indicating the kernel upload. This is immediately followed by:

```
Downloading other image size 0x26935d to 0xc0800000
```

followed by 14 lines of dots indicating the RAMdisk upload. Once the upload is complete whatever you specified on the exec line is executed.

Obviously the sizes and number of lines of dots and stars vary with the size of the uploaded images and the target addresses are what you specify in .angelrc.

The device should now be running, booting the kernel and using the RAMdisk as the root filing system. If you specified an exec line to run your serial program then it will start and show you the kernel boot info and login prompt.

Chapter 7. Tools and Techniques

7.1. Patching

This is the process of modifying a set of files with a 'patch' file containing a list of differences between two sets of files. This is how changes to source code are normally distributed. Patch files are generated by the **diff** command. You need at least version 2.5 of patch for patching the kernel. Here we use patching the binutils source as an example.

A patch is normally downloaded as a gzipped patch file (denoted by it ending in .gz). If so, gunzip it:

gunzip binutils-patch.gz

You can now patch binutils. Go into your unpacked binutils source directory, and run patch with your downloaded patch file, with argument `-p1`:

cd /usr/src/binutils-version
patch -p1 < binutils-patch

If you aren't sure exactly how a patch should be applied then the --dry-run option is very useful. Adding this means that patch will do its work, giving all the usual output but no changes are made to the files. To check out the effects of the above patch command, for example, you do:

patch -p1 --dry-run < binutils-patch

You should see a plethora of messages like:

```
Hunk #1 succeeded at 112
```

After patching has finished, you can check whether things have worked by searching for files ending in .rej (rejected patches):

% find . -name '*.rej'

In theory, there should be none of these files; all of the patches should have succeeded. However, if there are one or two failed files, it doesn't necessarily mean it won't build especially if the failed files are documentation or have names like `README` or endings like .info or .texi.

If you have several of these files, however, it could indicate something seriously wrong, and that it won't build. Check that the version of the source the patch is intended to be applied to matches the version of code you have. Also sometimes patches are 'reversed' - i.e. they have the original and modified source version the

wrong way round in the diff command that generated the patches. Patch is smart and if it looks like this is the case it will offer to do them all 'backwards' for you.

The -p1 in the above example refers to how many directory '/' separators patch should strip off the filenames before patching. This is to allow for differences in the depth of nesting between the source tree the patch was created on and the one it is being applied to. Normally you will find that -p1 or -p0 is correct. If a patch doesn't seem to work then looking at the depth of filename nesting in it and using a different -p option is the first thing to try.

Once you understand how this works you can save some commands by combining the unzipping and patching commands into one line like this:

zcat binutils-patch.gz | patch -p1

or, if the patch you have is bzipped instead of gzipped, then do:

bzcat binutils-patch.bz2 | patch -p1

7.2. Compiling a Kernel

7.2.1. Obtaining the Kernel Source

To compile your own kernel you need the source and the right set of patches. For all cases you need the base source tree and the ARM kernel patch. The SA1100 patches are usually a good idea too, and are necessary for SA1100 or later processors. For some devices, especially newly-supported ones, you may need a further patch for that device.

Kernel Source

Base source tree

`linux-2.4.1-tar.bz2`

or `linux-2.4.1-tar.gz`

ARM kernel (Russell King) patch

`patch-2.4.1-rmk1.gz`

SA1100 (Nico Pitre) patch

`diff-2.4.0-rmk1-np2.gz`

Each applied patch adds a suffix to the kernel name, so a kernel made with the 2.4.1 source with the rmk1 ARM kernel patch and the np2 SA1100 patch would be called zImage-2.4.1-rmk1-np2. The letters indicate the patch source. In the above cases they are the names of the patch maintainers Russell King and Nico Pitre respectively.

> **Note:** Although they use different naming conventions, both patch files are of the same type.

The kernel source is normally available in both gzipped and bzip2 form. The latter is significantly smaller so that is normally the better one to download. You will need the bzip2 utility to decompress bzip2 files.

All these are on the CD but you will need to copy them into your development directory. Alternatively, obtain the latest version of kernel base, ARM kernel and SA1100 patches from these ftp sites but remember to use your local mirror, e.g. ftp://ftp.uk.kernel.org/ for the UK mirror.

Kernel Source - Current Version

Kernel Base

ftp://ftp.kernel.org/pub/linux/kernel/v2.4/

ARM kernel Patch

ftp://ftp.arm.linux.org.uk/pub/linux/arm/source/kernel-patches/v2.4/

SA1100 Patch

ftp://ftp.arm.linux.org.uk/pub/linux/arm/people/nico/

The source tree can then be unpacked into your development source directory. You can put this anywhere you like, but it must be kept separate from your native source tree in /usr/src/linux. This is because many systems expect the version of headers in /usr/src/linux to correspond to the version of the kernel and glibc being run on the host machine. This is usually different from the version you are using for ARMLinux development. We recommend putting your ARM kernel source tree in `/usr/arm-linux/linux` (this is alongside the toolchain in a cross-compilation environment).

> **Note:** You need a fair amount of disk space free. The unpacked 2.4.1 source is a little over 100MB, and you should have two copies of it, and configuring and compiling only makes it bigger.

In your development directory enter the following command to unpack the source:

tar -xzvf linux-2.4.1.tar.gz

Or if you have the bzipped version, use these options:

tar -xIvf linux-2.4.1.tar.bz2

Finally, unzip/untar the source before applying the ARM and SA1100 patches to give you a new kernel source tree. Using patch -p0 can confuse various versions of patch so it's best to **cd** into the linux directory and use p1, thus:

cd linux
zcat ../patch-2.4.0-rmk1.gz | patch -p1
zcat ../diff-2.4.0-rmk1-np2.gz | patch -p1

7.2.1.1. Kernel source management

If you are intending to work on the kernel, the first thing you should do is take a copy of the pristine tree. This is so that you can generate patches against it after you have made changes. The simplest way to do this is to do:

cp `-a` linux linux-2.4.1.orig

However, as in practice you will only ever change a few files and have two copies of the whole thing, 99% of which are going to remain untouched, this means that you may prefer to use hard links to create the copy as then only the changed files take up space. This method relies on the editor creating a new file after every save so that the original is preserved. If you have need of such an editor, take a look at MicroEmacs or GNU Emac's *back-up-when-copying-when-linked* facility. To create a hard-linked source copy do this:

cp `-a -l` linux linux-2.4.1.orig

Alternatively, you can use CVS to control your source versions, but that is not covered here.

7.2.2. Making a Kernel

First you should tidy up the source to get rid of old files and stale dependencies, especially if you have previously configured for a different device with **make mrproper**. If you are not compiling the kernel natively, you must modify the top level kernel `Makefile`. You will need to change the line starting with `ARCH :=` to

`ARCH := arm` and the line with `CROSS-COMPILE =` to be the path and prefix to your compiler tools (e.g. `CROSS_COMPILE = /usr/bin/arm-linux-`). You should then issue the appropriate **make *target-device*_config** command to set up a default configuration file for your target device. The list of possibilities for *target-device* is found in `linux/arch/arm/def-configs/`. You will then need to set up a kernel configurations file using the defaults which have just been created. This can be accomplished with the **make oldconfig** command. Finally, **make dep** sets up kernel dependencies and sub-configuration files on the basis of the specified config whilst **make zImage** builds the kernel image. The kernel should consequently appear in `linux/arch/arm/boot/zImage`. So, to summarise:

make mrproper
make *target-device*_config
make oldconfig
make dep
make zImage

7.3. Making a RAMdisk

The RAMdisks that are available for popular devices are usually fine for getting started, but sooner or later you will need to make your own. The steps involved are:

1. Put all the files you want to put on the disk in a suitable directory with the correct directory structure;
2. Zero out a block of memory (this is so that the spare space in your RAMdisk compresses as much as possible - if it was full of whatever random stuff was in RAM at the time it would waste space in the final RAMdisk);
3. Make a filesystem in this memory;
4. Mount it;
5. Copy the files you prepared into it;
6. Unmount it;
7. Compress it.

So here's an example. The size of the RAMdisk is 2MB, your prepared files are in the directory `preparedfiles`, and we are making a conventional ext2 RAMdisk, rather than something more exotic. `/mnt/ramdisk` should already have been created as a mount point. You will need to be root.

dd if=/dev/zero of=/dev/ram bs=1k count=2048

mke2fs -vm0 /dev/ram 2048
mount -t ext2 /dev/ram /mnt/ramdisk
cp -av preparedfiles /mnt/ramdisk
umount /mnt/ramdisk
dd if=/dev/ram bs=1k count=2048 | gzip -v9 ramdisk.gz

The -m0 option to mke2fs specifies that no extra space for the super-user is reserved - again to minimise the size of the filesystem.

7.3.1. Looking at existing RAMdisks

To see what is in an existing RAMdisk you need to gunzip it and loop-mount it as a filesystem. Your kernel needs loop support for this to work, but a desktop kernel will normally have this (it's really useful). Let's say you have a RAMdisk called ramdisk_ks.gz (from the Assabet distribution). It's a good idea to work on a copy of the disk, not the original. To do this (`/mnt/ramdisk` should already have been created as a mount point and you will need to be root):

gunzip ramdisk_ks.gz
mount -o loop ramdisk_ks /mnt/ramdisk

Now you can see the contents by browsing /mnt/ramdisk like any other filesystem. You can even add files by copying them in if there is space left in the RAMdisk. The kernel needs to have support for the filesystem used in the RAMdisk.

7.4. Making your own patches

When you have changed some bit of the kernel, e.g. supported a new device or fixed a bug, you need to make a patch of your changes in order to distribute it to others and put it back in the kernel. This is easy to do. Say you have the patched version of the kernel in `linux` and the original version in `linux.orig`, then issue these commands.

make mrproper
diff `-urN` linux.orig linux > my_changes.diff

The **make mrproper** tidies up all the extra files lying around so that the **diff** is only of the actual relevant files.

> **Note:** The order of the `linux.orig` and `linux` arguments above is significant. If you do it the wrong way round you will have made a 'reversed' patch, which is at best confusing.

You now have `linux/` and `linux.orig/` as well as a patch - `my_changes.diff`. You may now want to rename `linux/` to `linux.orig` so that future patches do not duplicate the changes thus far - this enables each patch to be specific to a change, rather than monolithic. This helps the kernel maintainer (see below).

Russell King maintains an automated 'Patch State System' on his web site at http://www.arm.linux.org.uk/developer/patches/ which tracks patches applied to the ARMLinux kernel. It is intended that all ARMLinux developers who wish their patch to be integrated into the ARMLinux kernel (aka -rmk kernels) should submit their patch to this system. These patches will then find their way to Linus.

However, this does not give the full story of patches applied; changes that go into Linus' kernels are outside the scope of this patch system.

> **Note:** By submitting a patch or code to the patch system, you implicitly agree that it is suitable for redistribution with the kernel source. In other words, it is up to you to ensure that the material you submit to the patch system is suitable for inclusion in my kernel, and Linus' kernel. If you have any doubts about this point, please discuss it with Russell in private email.

There are certain restrictions on the system, so please follow these instructions carefully.

7.4.1. Sending in a patch

If you wish to send in a patch, please use the following procedure. It contains some important points that need to be followed:

1. Generate the patch:
   - You need to think about the content of the patch. It should cover one feature, one driver, or one bug fix only. If you are changing an existing file in the kernel, it's best to send this change as a separate patch. Putting closely related stuff into one patch is acceptable (e.g. a driver, its makefile changes and its config.in changes). If you are submitting the code for a new machine type, then putting the arch/arm/mach-* and include/asm-arm/arch-* changes in one large patch makes sense.
   - Try to make sure your patch will still apply against later kernel versions.

- Please tell diff to generate unified patches, recursing and new files if necessary. The options are **diff -urN**. You may also like to use the -p option so that diff includes the C function name that the change is occurring in.
- The diff should be generated using a path relative to the directory directly above the main linux/ source tree. Both the old and new directories must contain the same number of slashes otherwise patch gets confused. If you only want a small area of the kernel, then use something like diff -urN linux.org/arch/arm/kernel linux/arch/arm/kernel. In the extremely rare case that you absolutely must generate the patch using a different path, please make this obviously visible in the textual part of the patch mail (not following this will result in delays with application of your patch).

2. Give the mail a descriptive subject line. This will appear on the web page, and in the release notes for the next -rmk version of the kernel.
3. Include some text in the message explaining what the new feature is, the bug, or why the patch is needed.
4. Put on a blank line "PATCH FOLLOWS". There must be no space before or after these words on this line.
5. On a separate line, add a tag "KernelVersion: " followed by the kernel version that the patch was generated against.
6. Append your patch as plain unwrapped text after this line to the end of the message. Note that the patches must be precisely what diff generates. It is not acceptable for TABs to be converted to spaces, or extra newlines or spaces to be added into the file. If you are unsure about the behaviour of your mailer, send the patch to yourself and examine it.
7. Mail it to <patches@arm.linux.org.uk>

When you receive a reply, you may wish to supply a detailed follow up article that explains exactly what your patch does. You can do this by replying to the mail you receive from the patch system.

If at any other time you wish to follow up on the patch, please use the subject line as the key (it should include the exact string "(Patch #number)", where ’number’ is the patch number that you wish to follow up to).

Please absolutely do not send MIME encoded emails, even quoted-printable MIME encoded emails to the above address. If quoted-printable emails are sent, then you will receive an error message. All mails to <patches@arm.linux.org.uk> should be plain unwrapped text.

7.4.2. Advice

- If you have developed a device driver that uses a major/minor pair, please get it allocated according to the linux/Documentation/devices.txt file. Major/minor numbers are device driver specific, not device specific. Any device driver which re-uses major/minor numbers without an extremely good reason (included along with the patch) do not stand any chance of being accepted.
- To ease the problems of applying patches, I recommend that you send a patch that adds exactly one feature or bug fix in a single email. Mails containing zero or less, or two or more features stand a greater chance of not being applied if any one part of the patch is not acceptable for some reason.
- Patches that are inter-dependent should not be sent - these significantly increase the probability of both patches being rejected, unless both get applied at the same time. My recommendation is: don’t do it.
- All patches should be complete, that is applying it should not cause any breakage. i.e., adding a new architecture inline function in include/asm-arm/arch-* should be done such that all existing architectures will still at the very least compile correctly. If you’re not sure how it should be handled on a particular architecture, put in a GCC #warning statement.

Chapter 8. The GNU Toolchain

This chapter contains information on building a GNU toolchain for ARM targets.

8.1. Toolchain overview

The toolchain actually consists of a number of components. The main one is the compiler itself gcc, which can be native to the host or a cross-compiler. This is supported by *binutils*, a set of tools for manipulating binaries. These components are all you need for compiling the kernel, but almost anything else you compile also needs the C-library glibc. As you will realise if you think about it for a moment, compiling the compiler poses a bootstrapping problem, which is the main reason why generating a toolset is not a simple exercise.

This chapter contains information on how the toolchain fits together and how to build it. However, for most developers it is not necessary to actually do this. Building the toolchain is not a trivial exercise and for most common situations pre-built toolchains already exist. Unless you need to build your own because you have an unusual situation not catered for by the pre-built ones, or you want to do it anyway to gain a deeper understanding, then we strongly recommend simply installing and using a suitable ready-made toolchain.

8.2. Pre-built Toolchains

This CD contains three pre-built toolchains: one from emdebian.org, one from the LART project and a third from Compaq’s handhelds.org team. The emdebian chain is newest, and we’ve had good success with it, but all are used by various people. They all have very similar functionality.

At the moment the most likely situation where a pre-built toolchain will not do the job is if you need Thumb support, in which case you need to use gcc v3 (not yet released at the time of writing, but available as snapshots).

8.2.1. Native Pre-built Compilers

For binary versions of native compilers (i.e. ones that run on ARM and compile for ARM), the current stable release is on the Aleph ARMLinux CD. You can also get them from:

8.2.1.1. Resources

- The current stable release on Debian's master FTP site (armv3l and above) (ftp://ftp.debian.org/debian/dists/stable/main/binary-arm/devel/).
- The latest release on Debian's master FTP site (armv3l and above) (ftp://ftp.debian.org/debian/dists/unstable/main/binary-arm/devel/).

Sometimes ARMLinux.org will have experimental versions available.

8.2.2. Emdebian

The emdebian version includes gcc version 2.95.2, binutils 2.9.5.0.37, and glibc 2.1.3. Installation is simple. If you have a Debian system then the emdebian cross-compiler is incredibly simple to install - just show apt the directory on the CD and do **apt-get install task-cross-arm**. What could be simpler?

> **Warning**
>
> The emdebian cross development environment will install files in `/usr/bin` so you will have to make sure that you do not overwrite any development tools which you may already have on your system.

8.2.2.1. Installing the Toolchain

task-cross-arm is a *virtual* package which includes:

1. gcc version 2.95.2;
2. binutils 2.9.5.0.37;
3. glibc 2.1.3.

This is made up of the following packages:

1. the C preprocessor: cpp-arm_2.95.2-12e4_i386.deb;
2. the C compiler: gcc-arm_2.95.2-12e4_i386.deb;
3. the C++ compiler: g++-arm_2.95.2-12e4_i386.deb;
4. gnu C library: libc6-dev-arm_2.1.3-8e4_i386.deb;
5. C++ library: libstdc++2.10-arm_2.95.2-12e4_i386.deb;
6. C++ library and headers: libstdc++2.10-dev-arm_2.95.2-12e4_i386.deb;

7. Binary utilities: binutils-arm_2.9.5.0.37-1e3_i386.deb.

They are available in both DEB and RPM form.

In order to set up your cross development environment on a Debian system, proceed as follows:

- **su** to root by typing **su** at the prompt;
- add the line

  `deb http://www.emdebian.org/emdebian unstable main`

  to your `/etc/apt/sources.list` file;
- type **apt-get update** to tell apt/dselect to note the new packages available;
- enter **apt-get install task-cross-arm** to install your new development environment;
- type **exit** to become a normal user and begin to cross-compile.

For the RPM form download it and use:

rpm -i g++-arm-1%3a2.95..2-12e4.i386.rpm

8.2.3. LART

The LART tarball contains:

1. gcc 2.95.2;
2. binutils 2.9.5.0.22;
3. glibc 2.1.2.

8.2.3.1. Installing the Toolchain

In order to install the LART tarball on your system, do the following:

mkdir /data
mkdir /data/lart
cd /data/lart
bzip2 `-dc` arm-linux-cross.tar.bz2 | **tar** `xvf -`

You can then add `/data/lart/cross/bin` to your path. The C and C++ compilers can then be invoked with **arm-linux-gcc** and **arm-linux-g++** respectively.

8.2.4. Compaq

The Compaq arm-linux cross toolchain includes:

1. gcc-2.95.2;
2. binutils-2.9.5.0.22;
3. glibc-2.1.2 with the international crypt library.

The toolchain is compiled with a i386 host with an armv41 target.

8.2.4.1. Installing the Toolchain

> **Note:** The toolchain must be installed in `/skiff/local` as it will not work from any other path.

The only other problem that you may have with the include files is that the tarball was set up for Linux 2.2.14. You may consequently need to set up symbolic links:

ln `-s` /usr/src/linux/include/asm
/skiff/local/arm-linux/include/asm
ln -s /usr/src/linux/include/linux
/skiff/local/arm-linux/include/linux

Alternatively, copy `/usr/src/linux/include/asm` and `/usr/src/linux/include/linux` to `/skiff/local/arm-linux/include` before running **make menuconfig** and **make dep**. This will verify that your kernel tree and correct symbolic links are up to date.

> **Note:** This toolchain has glibc symbol versioning. If you are using a NetWinder, you may have to compile your code with static libraries.

8.3. Building the Toolchain

In outline what you need to do is:

- decide on the target name;

- decide where to put the images;
- work out headers policy;
- compile binutils first;
- then compile gcc;
- produce gLibc last.

8.3.1. Picking a target name

A native compiler is one that compiles instructions for the same sort of processor as the one it is running on. A cross-compiler is one that runs on one type of processor, but compiles instructions for another. For ARM embedded development it is common to have a compiler that runs on an x86 PC but generates code for the target ARM device.

What type of compiler you build and the sort of output it produces is controlled by the 'target name'. This name is what you put in instead of 'TARGET-NAME' in many of the examples in this chapter. Here are the basic types:

arm-linux

This is the most likely target you'll want. This compiles ELF support for *Linux/ARM* (i.e. standard ARMLinux). ELF is the best and most recent form for binaries to be compiled in, although early Acorn Linux/ARM users may still be using the old 'a.out' format.

arm-linuxaout

This produces Linux/ARM flavour, again, but using the 'a.out' binary format, instead of ELF. This is older, but produces binaries which will run on very old ARMLinux installations. This is now strongly deprecated; there should be no reason to use this target; note that binutils 2.9.5 doesn't contain support for it (refer to Section 8.3.6.3).

arm-aout, arm-coff, arm-elf, arm-thumb

These all produce *flat*, or standalone binaries, not tied to any operating system. arm-elf selects Cygnus' ELF support, which shares much of its code with arm-linux.

You can fiddle with the *arm* bit of the target name in order to tweak the toolchain you build by replacing it with any of these:

armv2

This makes support for the ARM v2 architecture, as seen in the older ARM2 and ARM3 processors. Specifically, this forces the use of 26-bit mode code, as this is the only type supported in the v2 architecture.

armv3l, armv3b

This makes support for the ARM v3 architecture, as seen in the ARM610, and ARM710. The *l* or *b* suffix indicates whether little-endian or big-endian support is desired (this will almost always be little-endian).

armv4l, armv4b

This makes support for the ARM v4 architecture, as used in the StrongARM, ARM7TDMI, ARM8, ARM9.

armv5l, armv5b

This makes support for the ARM v5 architecture, as used in the XScale and ARM10.

In practice the target name makes almost no practical difference to the toolchain you get anyway so you should always use plain 'arm'. This means that the toolchain itself is not unhelpfully tied to the type of processor it was built on. You get a toolchain that will run on all ARM processors and you can set the compiler output for the target processor when you build each part of the toolchain.

8.3.2. Choosing a directory structure

In many of the shell commands listed in this document you'll see italicised and emboldened bits of text. These are, on the whole, directory paths which will change depending on exactly how you've configured your toolchain. This means that we have not used an actual directory path in examples as it could be different from your setup. You need to substitute the correct value for your setup yourself for any commands that we have listed in this document.

Here is a list of these items:

PREFIX

This is the base directory containing all the other subdirectories and bits of your toolchain; the default for the native toolchain on any system is almost always `/usr`. To keep from stepping on your system's native tools when you build a cross-compiler you should put your cross-development toolchain in `/usr/local`, or `/usr/arm/tools` or somewhere else that makes sense for you, in order to keep it separate and easy to maintain.

TARGET-PREFIX

If you're building a cross-toolchain, this is equal to PREFIX/TARGET-NAME (e.g. `/usr/arm_tools/arm-linux`). If you're building a native compiler, this is simply equal to PREFIX.

KERNEL-SOURCE-LOCATION

This is the place where your kernel source (or at least headers) are stored. Especially if you are cross compiling this may well be different to the native set of files. We recommend that you set this to TARGET-PREFIX/linux as a sensible default.

8.3.3. Binutils

Binutils is a collection of utilities, for doing things with binary files.

8.3.3.1. Binutils components

addr2line

Translates program addresses into filenames and line numbers. Given an address and an executable, it uses the debugging information in the executable to figure out which filename and line number are associated with a given address.

ar

The GNU **ar** program creates, modifies, and extracts from archives. An archive is a single file holding a collection of other files in a structure that makes it possible to retrieve the original individual files (called members of the archive).

as

GNU **as** is really a family of assemblers. If you use (or have used) the GNU assembler on one architecture, you should find a fairly similar environment when you use it on another architecture. Each version has much in common with the others, including object file formats, most assembler directives (often called pseudo-ops) and assembler syntax.

as is primarily intended to assemble the output of the GNU C compiler gcc for use by the linker ld. Nevertheless, we've tried to make **as** assemble correctly everything that the native assembler would. This doesn't mean **as** always uses the same syntax as another assembler for the same architecture.

c++filt

: The **c++filt** program does the inverse mapping: it decodes (demangles) low-level names into user-level names so that the linker can keep these overloaded functions from clashing.

gasp

: Gnu Assembler Macro Preprocessor.

ld

: The GNU linker **ld** combines a number of object and archive files, relocates their data and ties up symbol references. Often the last step in building a new compiled program to run is a call to ld.

nm

: GNU **nm** lists the symbols from object files.

objcopy

: The GNU **objcopy** utility copies the contents of an object file to another. **objcopy** uses the GNU BFD library to read and write the object files. It can write the destination object file in a format different from that of the source object file. The exact behaviour of **objcopy** is controlled by command-line options.

objdump

: **objdump** displays information about one or more object files. The options control what particular information to display.

ranlib

: **ranlib** generates an index to the contents of an archive, and stores it in the archive. The index lists each symbol defined by a member of an archive that is a relocatable object file. You may use ‘nm -s’ or ‘nm --print-armap’ to list this index.

readelf

: **readelf** Interprets headers on elf files.

size

: The GNU **size** utility lists the section sizes and the total size for each of the object files objfile in its argument list. By default, one line of output is generated for each object file or each module in an archive.

strings

GNU **strings** prints the printable character sequences that are at least 4 characters long (or the number given with the options below) and are followed by an unprintable character. By default, it only prints the strings from the initialized and loaded sections of object files; for other types of files, it prints the strings from the whole file.

strip

GNU **strip** discards all symbols from the target object file(s). The list of object files may include archives. At least one object file must be given. **strip** /modifies the files named in its argument, rather than writing modified copies under different names.

8.3.3.2. Downloading, unpacking and patching

The first thing you need to build is GNU binutils. 2.9.5 versions have proved stable but generally the latest release is recommended (2.10.1 at the time of writing). No-one should be using 2.9.1 anymore.

Download the latest version you can find from any of these sites:

- ftp://ftp.gnu.org/gnu/binutils/" - the official release site (US); (ftp://ftp.gnu.org/gnu/binutils/)
- H. J. Lu's own site -- this has the very latest stuff (US); (ftp://ftp.varesearch.com/pub/support/hjl/binutils)
- src.doc.ic.ac.uk (UK). (ftp://src.doc.ic.ac.uk/Mirrors/sunsite.unc.edu/pub/Linux/GCC)

Unpack the archive somewhere handy, like `/usr/src`:

```
cd /usr/src
tar -xzf /.../binutils-2.10.1.tar.gz
```

There may be ARM-specific patches available for binutils which resolve various bugs, or perhaps improve performance; it's usually a good idea to apply these to the source, if they exist. However ARM binutils are now quite stable (at the time of writing) and integrated with the main tree, so extra patches are no longer normally required. The best place to get up to date information is the armlinux-toolchain mailing list.

The place to check if there are any currently recommended patches is here:

- ftp://ftp.armlinux.org/pub/toolchain (UK). (ftp://ftp.armlinux.org/pub/toolchain)

If you do need to patch binutils refer to Section 7.1 for details of the procedure.

8.3.3.3. Configuring and compiling

Essentially, you want to follow the instructions provided in the file called `INSTALL`. In practice you'll probably use one of the following examples.

If you're building a native toolchain, i.e. you're building on an ARM machine for an ARM machine, then you should just do this from inside the binutils directory:

./configure --prefix=PREFIX

If you're building on another machine (such as an x86 Linux box), and you want to build a cross-compiler for the ARM, try this:

./configure --target=TARGET-NAME --prefix=PREFIX e.g. **./configure --target=arm-linux --prefix=/usr/arm_tools**

This should succeed (i.e. proceed without stopping with anything that looks like an obvious error message), and you can then actually start the build.

Invoke **make** in the binutils directory:

make

This should proceed without incident.

If it works, you can then install your new binutils tools, making sure you've read the overwriting warning below (refer to Section 8.3.6.2):

make install

You'll notice your new set of tools in PREFIX/TARGET-NAME/. One likely location may be in `/usr/arm_tools/arm-linux/`.

Right, we're done with binutils. Now we move on to the compiler.

8.3.4. gcc

8.3.4.1. Kernel headers

> **Note:** When building a native compiler, most likely a set of kernel headers for your platform will be available and you don't need to be concerned with headers. For cross-compiling, a set of kernel headers from a source tree configured for your target must be available.

The overwhelming chances are that KERNEL-SOURCE-LOCATION for a native compiler build will be `/usr/src/linux`. Now skip the rest of this section.

However if you are compiling for a different type of ARM machine, or compiling a different version of the kernel or cross-compiling, then you need a different set of headers. First off we need to get hold of a current Linux/ARM kernel. Download the latest kernel archive you can find (version 2.4.1 at the time of writing):

- ftp.uk.kernel.org (UK) (ftp://ftp.uk.kernel.org/pub/linux/kernel/)
- ftp.kernel.org (US) (ftp://ftp.kernel.org/pub/linux/kernel)

We recommend you use a version 2.4 kernel (i.e. one in the v2.4 directory). There are several reasons for this; many newer ARM architectures are only really properly supported in kernel 2.4 and development on version 2.0 has ceased, whilst 2.2 is now in maintenance mode. However, 2.2 kernels are significantly smaller, so if it has the functionality you need it may make sense to use one, but you will be running against the flow to an increasing extent.

Unpack this somewhere, although preferably not in `/usr/src`. If you're on a Linux system, the chances are you'll trash whatever Linux kernel source you already have installed on your system. We suggest /usr/PREFIX/ (e.g. /usr/arm_tools/arm-linux/).

There are a wide variety of patches you can apply to Linux kernel source for ARM. Applying the kernel patches tends to be mandatory. The two basic patches we recommend are:

- the latest patch you can find on ftp.arm.linux.org.uk (ftp://ftp.arm.linux.org.uk/pub/armlinux/source/kernel-patches) for your version of the kernel (currently `patch-2.4.1-rmk1.gz for version 2.4`);
- the latest patch you can find in Nicolas Pitre's StrongARM patches (ftp://ftp.netwinder.org/users/n/nico/) for your version of the kernel (currently `diff-2.4.1-rmk1-np2.gz` for version 2.4).

You may possibly require patches for specific hardware (e.g. iPAQ, Psion), but this is unlikely for here: we are only trying to get the kernel headers into a state where they can be used to compile gcc; we don't have to worry about device driver support and so forth.

Apply these two patches as shown in Section 7.1, in sequence (assuming you want to use both of them).

Now you need to specify that you want to build for the ARM architecture. You can either set this permanently in the top-level `Makefile` (convenient if you only ever build for one architecture), or specify it on the **make** command line.

To change the `Makefile` look for a line like this:

```
ARCH := $(shell uname -m | sed -e s/i.86/i386/ -e
s/sun4u/sparc64/ -e s/arm.*/arm/ -e s/sa110/arm/)
```

Delete it, or comment it out, and insert this:

```
ARCH = arm
```

To specify the architecture on the command line you append `ARCH=arm` to each kernel make command so **make menuconfig** becomes **make menuconfig ARCH=arm**. This has the advantage of not changing the kernel source and you can easily compile for other architectures in the same way.

Next you need to configure the kernel, even though you won't necessarily want to compile from it. So change to the top-level directory of the kernel source and do:

make menuconfig (or **make menuconfig ARCH=arm**)

Go into the top option: 'System and processor type', and select a system consistent with the tools you're building.

For example, if you're building a set for arm-linux with a StrongARM that doesn't need to support arm v3, select 'LART'. In practice, the toolchain only takes a couple of items from the kernel headers so it doesn't actually matter much what you select, so long as you select something and the correct links are made so that the files needed are visible in the right places.

Exit the configuration program, tell it to save the changes, and then run:

make dep ARCH=arm (or **make dep ARCH=arm**)

This command actually makes the links (linking /linux/include/asm/ to /linux/include/asm-arm etc) and ensures your kernel headers are in tip-top condition for the toolchain.

Having patched up your kernel appropriately, you are ready to go if you are building a cross-development toolchain. If you are doing a native toolchain build, however, you will have to copy the headers across into your new toolchain's directory:

mkdir TARGET-PREFIX/include

cp -dR KERNEL-SOURCE-LOCATION/include/asm-arm TARGET-PREFIX/include/asm

cp -dR LINUX-SOURCE-LOCATION/include/linux TARGET-PREFIX/include/linux

Now gcc will have its headers, and compile happily.

8.3.4.2. Downloading, unpacking and patching gcc

Download the latest version (2.95.3-prerelease at the time of writing), unless you need thumb support or are feeling brave, in which case you can try a CVS snapshot of the forthcoming v3.0 from any of these sites:

- gcc.gnu.org (US) (ftp:///gcc.gnu.org/)
- sourceware.cygnus.com Mirror (UK) (ftp://sunsite.doc.ic.ac.uk/Mirrors/sourceware.cygnus.com/pub/gcc/)

We suggest you grab `gcc-core-2.95.3.tar.bz2` (8MB) (in the `gcc-2.95.3` directory on the gcc site), or even `gcc-2.95.3.tar.bz2` (12MB) if you're feeling masochistic and want the whole thing.

Following through the same instructions as above, unpack your downloaded gcc. Then you may choose to apply patches if they exist. As of gcc 2.95.2 patches for ARM are not generally required. Check out the armlinux-toolchain list archives for the current state of play or look here to see if there are any current patches for your version:

- ftp://ftp.armlinux.org/pub/toolchain (UK). (ftp://ftp.armlinux.org/pub/toolchain)

8.3.4.3. Configuring and compiling

You can now configure gcc in a similar way that you did for binutils (reading `INSTALL` as you go).

Most people will have arm-linux as the target name they're configuring for, which builds a toolchain suitable for running on any ARM. For a native compiler do this:

./configure --prefix=PREFIX

For a cross-compiler, do this::

./configure --target=TARGET-NAME --prefix=PREFIX
--with-headers=LINUX-SOURCE-LOCATION/include

e.g. **./configure --target=arm-linux --prefix=/usr/arm_tools**
--with-headers=/usr/src/linux/include

Configuring done, now we can build the C compiler portion.

This is probably the trickiest stage to get right; there are several factors to consider:

- Do you have a fully-working and installed version of glibc *for the same ABI* as that for which you are building gcc? (i.e. is your existing glibc for the same processor-type and binary format etc). If this is your first time building a cross-compiler, then the answer is almost certainly no. If this is not your first time

building, and you built glibc previously, in the same format as you're using for gcc now, then the answer might be yes.

If the answer is no, then you cannot build support for any language other than C, because all the other front-ends depend on libc (i.e. the final gcc binary would expect to link with libc), so if this is your first build, or you changed to a different target, then you must add the switches **--enable-languages=c --disable-threads** to the gcc configurations listed above.

- Do you even have the libc headers for your target? If this is the very first time you have built a cross-compiler on your host, then the chances are that the answer is no. However, if you have previously successfully completed a compilation of a cross-compiling gcc, and installed it in a location that can be found this time round, the answer is probably yes.

  If the answer is no, you will probably need to employ the "Dinhibit_libc" hack (refer to Section 8.3.6.4); however, it's worth attempting a build first to see whether you're affected or not. (Most likely you will be if this is your first cross-compile attempt).

Invoke make inside the top-level gcc directory, with your chosen parameters. The most common error looks like this:

```
./libgcc2.c:41: stdlib.h: No such file or directory
./libgcc2.c:42: unistd.h: No such file or directory
make[3]: *** [libgcc2.a] Error 1
```

and is common with first time building of cross-compilers (see above). You can fix it, this time, using the -Dinhibit_libc hack (refer to Section 8.3.6.4) -- follow through the instructions in the hack, and then restart at the configure stage.

The other relatively likely error is if you have not been consistent about the --prefix and --target options between configuring binutils and configuring gcc. They have to be the same. In this case you will get an assembler error complaining that some ARM assembly is not recognized by the x86 assembler, looking something like this:

```
/tmp/ccSr9uGs.s: Assembler messages:
/tmp/ccSr9uGs.s:72: Error: no such 386 instruction: ‘dividend .req r0’
/tmp/ccSr9uGs.s:73: Error: no such 386 instruction: ‘divisor .req r1’
```

Assuming that worked, you can now install your spiffy new compiler:

make install

If the make terminated normally, congratulations. You (probably) have a compilation environment capable of compiling kernel-mode software, such as the Linux kernel itself. Note that when building a cross-compiler you will probably see error messages in the transcript saying that the compiler that was built doesn't work. This is because the test that's performed creates a small executable in the target, not the host format, and as a result will fail to run and generate the error. If you're only after cross-compiling kernels, feel free to stop here. If you want the ability to compile user-space binaries, press on.

8.3.5. glibc

glibc is the C library. Almost all userland applications will link to this library. Only things like the kernel, boot loaders and other things that avoid using any C library functions can be compiled without it. There are alternatives to glibc for small and embedded systems (it's a big library) but this is the standard and it's what you should use for compiling the rest of gcc.

8.3.5.1. Downloading and unpacking

glibc is split into bits (called add-ons):

- the *linuxthreads* code which is in a separate archive;
- the crypto stuff (which used to be an add-on) is now included in the latest release (glibc-2.2.2).

Fetch the main glibc archive (currently `glibc-2.2.2.tar.gz`) and the corresponding linuxthreads archive from one of the following:

- ftp.gnu.org/gnu/glibc (US) (ftp://ftp.gnu.org/gnu/glibc);
- ftp.funet.fi (Finland) (ftp://ftp.funet.fi/pub/gnu/funet);

Unpack the main glibc archive somewhere handy like `/usr/src`. Then unpack the two add-on archives inside the directory created when you unpacked the main glibc archive. All set.

8.3.5.2. Configuring and compiling

This is slightly more complicated than the previous section. The most important point is that before doing any configuring or compiling, you must set the C compiler

that you're using to be your cross-compiler, otherwise glibc will compile as a horrible mix of ARM code and native code. This is specified by the CC system variable. Do this in the same shell you're going to compile in:

CC=TARGET-NAME-gcc

Be sure to add the path to **TARGET-NAME-gcc** to your PATH environment variable as well. Create a new directory next to the top level source directory for gcc. Go into this directory, and configure and build glibc here. It is a very bad idea to configure in the glibc source directory (see the README file for further warnings). We won't detail the reasons here. Now we can configure glibc. Go into the top-level glibc directory, and you'll probably want to run configure more or less like this:

../glibc-2.2.2/configure arm-TARGET-NAME --build=NATIVE-TARGET --prefix=TARGET-PREFIX --enable-add-ons

So what do all the variables mean? arm-TARGET-NAME is important: at present the glibc configuration scripts don't recognise the various mutations of the *arm-* bit of the target name. So here you have to specify your normal target name, but changing the first arm- bit back to simply arm, rather than, say, armv3l.

NATIVE-TARGET is the target name of the machine you're building on; for instance on an x86 Linux machine, i586-linux would probably do nicely.

You'll notice the prefix is different this time: not just PREFIX, but with the target name component on the end as well.

> **Warning**
>
> Don't forget to specify this last component, or you may hose your local libraries, and thus screw up your system.

Okay, go ahead and configure, reading `INSTALL` if you want to check out all the options. Assuming that worked, just run:

make
make install

And if *those* worked, you're sorted. You have a full ARM toolchain and library kit available for use on your system - however it can only compile C programs. To be able to compile anything else, you need to do a bit more work.

Go back and re-configure gcc but this time either add the languages you want (e.g. **--enable-languages=c,c++** or else leave off the switch altogether to build everything. You must also remove the "Dinhibit_libc" hack if you had to apply it before. WARNING: be sure to unset the 'CC' environment variable when

cross-compiling so the native compiler will be used to finish building your cross-development tool chain.

You can now try compiling things by using TARGET-NAME-gcc (e.g arm-linux-gcc) as your compiler; just set CC as above (e.g CC=arm-linux-gcc) before compiling any given package, and it should work. For any package using configure, you can normally set CC in the makefile, rather than as a local system variable. Setting the local system variable can be a problem if later on you need to compile something natively and you forget to unset it. Be sure that a path to all of the toolchain binaries exists in your PATH environment variable.

8.3.6. Notes

8.3.6.1. libgcc

Whatever target name you build gcc for, the main code engine still contains support for all the different ARM variations (i.e. it's the same whatever target name you build with). However, there is a library accompanying gcc, containing some fundamental support routines, called `libgcc.a`. This is the thing that will differ between different target names, and this is what makes different toolchains binary incompatible.

> **Note:** Exactly the same incompatibilities apply to glibc as well.

8.3.6.2. Overwriting an existing toolchain

If you're building a native compiler, with a significantly different target from the current one, you must be aware that it is extremely easy to trash your toolchain half-way through building. The most common cause of this is trying to build a native set of ELF tools on a system where gcc was built to produce a.out code (e.g. older Linux/ARM systems running on Acorn hardware). This is no longer a significant issue, but the example remains valid. The crucial breaking point is the **make install** command which installs binutils. In the example scenario, this will leave you unable to link any programs, as gcc's libraries will be in a.out format, but all your binutils will be unable to understand anything but ELF.

So, how does one get round this? A solution is to initially build everything with a "PREFIX" of something like `/usr/local/arm-tmp`, so as not to interfere with the existing toolchain. Then, go back and compile everything again, but using your proper prefix (e.g. `/usr`), but making sure `/usr/local/arm-tmp` (or whatever

you used) is on your $PATH environment variable. Then, having built everything in the correct directory, swipe `/usr/local/arm-tmp`.

If you do manage to trash your toolchain, you will need to go and fetch a suitable toolchain for your existing installation.

8.3.6.3. Issues with older version of binutils and gcc

There are some significant differences between binutils 2.9.1 and 2.9.5 and between gcc 2.95.1 and 2.95.2. If you use an old binutils with a new compiler or vice versa then things will go wrong. The indications of this areas follows: the error 'unrecognised emulation: armelf_linux' means your toolchain is too old for your compiler. Conversely 'unrecognised emulation: elf32arm' means your compiler is too old for your toolchain.

8.3.6.4. The -Dinhibit_libc hack

Upon installing a successful build of gcc, some headers will get put in the target's *include* directory. However, if you are building a (cross) compiler for the very first time, or with a different set of paths, it won't have these headers to hand. For the first time you build a gcc then, you can follow through these steps to fix the problem:

- Edit `gcc/config/arm/t-linux`

  and add

  `-Dinhibit_libc` and **`-D__gthr_posix_h`**

  to:

  `TARGET_LIBGCC2_CFLAGS`.

  That is, change the line that looks like:

  ```
  TARGET_LIBGCC2_CFLAGS = -fomit-frame-pointer -fPIC
  ```

  to:

  ```
  TARGET_LIBGCC2_CFLAGS = -fomit-frame-pointer -fPIC
  -Dinhibit_libc -D__gthr_posix_h
  ```
  .

- Re-run configure, but supplying the extra parameter `--disable-threads`.

8.4. Links

Other useful sources of information include:

- Phil Blundell's web site (http://www.tazenda.demon.co.uk/phil/). This web site is a good place to check for information that isn't out of date.
- The crossgcc FAQ is an excellent source of information on building cross-compilers, and should be read.
- Chris Sawer on kernel-compiling hints (http://members.xoom.com/chrissawer/armlinux.html)
- The main ARMLinux pages (http://www.arm.linux.org.uk/) often have useful information on them, and are a good starting point.

Chapter 9. Porting the Linux Kernel to a new ARM Platform

9.1. Introduction

So, you have a bit of ARM-based hardware that you want to port the Linux kernel to. This is a task that a competent software engineer can undertake assisted by relevant information such as this chapter, although previous familiarity with the Linux kernel will make it a lot easier. If your platform is a lot like something that has gone before then the port can be relatively simple, but if it's all new then it could be a big job, and you might well be advised to get help from someone experienced in these things, depending on how much of a challenge you want. And of course, if you don't actually know that your hardware works properly (you usually know this if it has already run some OS other than Linux), then again things can get hairy as you may not know if the hardware is broken or your kernel changes are wrong.

This chapter can't tell you everything you need to know about kernel hacking - it's a huge subject. If you don't know how the kernel works you need to read some relevant documentation. What I will try to cover are the procedures and conventions used in the ARM kernel development community, how the ARM architecture files are set out in the source, and the basics of what you will need to change in order to port the core of the kernel to your new platform - enough so that it boots and sends serial debug info. This should get you up to speed much quicker than having to work it all out yourself, and should help you to avoid asking stupid questions that have been asked hundreds of times before on the linux-arm-kernel mailing list.

The platform we will use in the examples is *anakin*, as this isn't too complicated, but is sufficiently unlike other platforms to be an architecture, rather than a sub-architecture, and thus be a non-trivial example. In case you were wondering, Anakin is a vehicle telematics platform designed by Acunia b.v, of Belgium, for which Aleph One did the initial kernel port. It contains a StrongARM 110 with some memory and an FPGA containing the core devices (memory controller, video controller, serial ports, interrupt controller).

9.2. Terminology

Talking about this subject can be confusing with multiple meanings and some

overlap of the terms ’device’, ’platform’, ’machine’, and ’architecture’, depending on context. For example, a device can be a thing such as a PDA, or a bit of hardware that the kernel must access via a device-driver. Here are the definitions used in this chapter.

1. *architecture*: Either the CPU-type (as in ’ARM architecture’, ’x86 architecture’), or category of system design as in ’footbridge architecture’ or ’sa1100 architecture’. It is sometimes used to mean the same as ’machine’ below too, but I try to avoid that usage here;
2. *device*: A hardware item that the kernel accesses using a device-driver;
3. *machine*: Your particular hardware, as determined by the assigned machine-ID within the kernel;
4. *platform*: Same as machine;
5. *sub-architecture*: Same as machine.

A machine may be a system architecture in its own right, but is usually a sub-architecture.

9.3. Overview of files

The starting point here is that you have set up your kernel source tree with the ARM patches. This description covers the 2.4 series kernels, specifically 2.4.18. Things will change as kernel development continues. The ARM-specific files are in `linux/arch/arm` (code), and `linux/include/asm-arm` (header files). In a configured kernel tree, the appropriate architecture headers directory (`linux/include/asm-arm` in our case) appears as `linux/include/asm` so that it can be easily referred to by other files that don’t need to know which architecture they are dealing with. Your machine-specific sub-directories go into these directories.

Device drivers for things, even if they are only found on the ARM architecture, or even only on your machine, do not go in these directories, but in the relevant parts of the main tree; usually `linux/drivers/`, but sometimes `linux/fs` or `/linux/net` if there are filesystems, partition types or network protocols specific to your device.

Within the ARM-specific directories your new or changed files go in appropriate `linux/arch/arm/mach-XXX` and `linux/include/asm-arm/arch-XXX` directories. e.g. `linux/arch/arm/mach-anakin` and `linux/include/asm-arm/arch-anakin`. After configuration your headers directory `linux/include/asm-arm/arch-XXX` appears as

`linux/include/asm-arm/arch` so that the correct machine header files can be included by using this path.

The other directories in `linux/arch/arm/` contain the generic ARM code:

- *`kernel`* - ARM-specific core kernel code;
- *`mm`* - ARM-specific memory management code;
- *`lib`* - ARM-specific or optimised internal library functions (backtrace, memcpy, io functions, bit-twiddling etc);
- *`nwfpe and fastfpe`* - two different floating-point implementations;
- *`boot`* - the directory in which the final compiled kernel is left and which contains stuff for generating compressed kernels;
- *`tools`* - has scripts for autogenerating files, such as `mach-types` (see Section 9.4);
- *`def-configs`* - contains the default configuration files for each machine.

The non machine-specific directories in `linux/include/asm-arm` are:

- *`arch`* - the link to the configured machine headers sub-directory arch-XXX;
- *`hardware`* - headers for ARM-specific companion chips or devices;
- *`mach`* - generic interface definitions for things used by many machines (irq, dma, pci) and the machine description macros;
- *`proc`* - link to proc-armo or proc-armv appropriate for configured machine;
- *`proc-armo`*, *and* *`proc-armv`* - 26 and 32-bit versions of core processor-related headers.

Not every new machine is a whole new architecture, most are only sub-architectures - e.g. all the SA1100 and SA1110-based devices are grouped together under the *sa1100* architecture in `linux/arch/arm/mach-sa1100`. You will need to take a look at the existing machines to see if yours more naturally goes into the tree as a sub-architecture or not. It is useful to do a bit of research to see which other machines are closest to yours in various aspects (e.g. memory map, companion IO chip, other devices). Often the easiest way to start is to copy over the files of the nearest machine to yours.

9.3.1. armo and armv

Throughout the ARM architecture core directories you will find both `-armv` and `-armo` versions of some files. These indicate variants for the ARM processors

26-bit mode and 32-bit mode. The 26-bit mode is in the process of being phased out of ARM CPU designs and is only used in a few early machines which predate the existence of the 32-bit mode (e.g. the A5000, which has an ARM3 CPU).

The suffixes have the following meaning:

1. *armo* is for 26-bit stuff;
2. *armv* is for 32-bit stuff.

9.4. Registering a machine ID

Each device is identified in the kernel tree by a *machine ID*. These are allocated by the kernel maintainer to keep the huge number of ARM device variants manageable in the source trees.

The first thing you need to do in your porting work is register your new machine with the kernel maintainer to get a number for it. This is not actually necessary to begin work, but you'll need to do this eventually so it's best to do it at the beginning and not have to change your machine name or ID later.

You register a new architecture by mailing <rmk@arm.linux.org.uk>, or filling in an on-line form at http://www.arm.linux.org.uk/developer/machines/. The on-line version is preferred as this will also set up the password you need to use the patch-system.

If using mail, please give the mail a subject of **`Register new architecture`**:

Name: <name of your architecture>
ArchDir: <name of include/asm-arm/arch-* directory>
Type: <MACH_TYPE_* macro name>
Description:
<description of your architecture>

Please follow this format - it is an automated system. You should receive a reply within one day like this:

```
> You have successfully registered your architecture!
>
>   The registered architecture name is
>      Anakin
>
>   The architecture number that has been allocated is:
>      57
```

```
>
>   This number corresponds to the following macros:
>      machine_is_anakin()
>      MACH_TYPE_ANAKIN
>      CONFIG_ARCH_ANAKIN
>
>   and your architecture-specific include directory is
>      include/asm-arm/arch-anakin
>
> If, in the future, you wish to alter any of these, entries, please
> contact rmk@arm.linux.org.uk.
```

Then you need to add the info to `linux/arch/arm/tools/mach-types` with a line like this:

```
 machine_is_xxx                          CONFIG_xxx
               MACH TYPE_xxx                       machine_ID
lart                   SA1100_LART          LART             27
anakin                 ARCH_ANAKIN          ANAKIN           57
```

or go to: http://www.arm.linux.org.uk/developer/machines/ where you can download the latest version of `mach-types`.

The above file is needed so that the script `linux/arch/arm/tools/gen-mach-types` can generate `linux/include/asm-arm/mach-types.h` which sets the defines and macros mentioned above that are used by much of the source to select the appropriate code. You should always use these macros in your code, and not test `machine_arch_type` nor `__machine_arch_type` directly as this will produce less efficient code. The macros are created in such a way that unused code can be efficiently optimised away by the compiler.

9.5. Config files

Add a new config file in `linux/arch/arm/def-configs/` named `<machine-name>`, containing the default configuration options for your machine. You should also edit `linux/arch/arm/config.in` so that **make config** will support your machine. This file specifies the new CONFIG_ symbols for your machine and the dependencies of them on other CONFIG_ symbols.

When you do **make <machine-name>_config**, e.g. **make anakin_config**, to build a kernel, then the file corresponding to the first part of the parameter is copied out of `linux/arch/arm/def-configs/` into `linux/.config`.

9.6. Kernel Basics

There are a number of basic symbols that you need to know the meanings of to understand the kernel sources. Here is a list of the most important ones.

Throughout the code you need to keep in mind the mapping between physical and virtual memory. The kernel deals exclusively in virtual memory once it has started. Your hardware specifications deal in physical memory. One of the fundamental things you need to specify is the mapping between these two. This is contained in the `__virt_to_phys()` macro in `include/asm-arm/arch-XXX/memory.h` (along with corresponding reverse mappings). Normally, this macro is simply:

```
phys = virt - PAGE_OFFSET + PHYS_OFFSET
```

9.6.1. Decompressor Symbols

ZTEXTADDR

Start address of decompressor. As the MMU is off at the time this code is called the addresses are physical. You normally call the kernel at this address to start it booting. This doesn't have to be located in RAM, it can be in flash or other read-only or read-write addressable medium.

ZBSSADDR

Start address of zero-initialised work area for the decompressor. This must be pointing at RAM. The decompressor will zero initialise this for you. Again, the MMU will be off.

ZRELADDR

This is the address where the decompressed kernel will be written, and eventually executed. The following constraint must be valid:

```
__virt_to_phys(TEXTADDR) == ZRELADDR
```

The initial part of the kernel is carefully coded to be position independent.

INITRD_PHYS

Physical address to place the initial RAM disk. Only relevant if you are using the bootpImage stuff (which only works on the older struct param_struct style of passing the kernel boot information).

INITRD_VIRT

Virtual address of the initial RAM disk. The following constraint must be valid:

__virt_to_phys(INITRD_VIRT) == INITRD_PHYS

PARAMS_PHYS

Physical address of the struct param_struct or tag list, giving the kernel various parameters about its execution environment.

9.6.2. Kernel Symbols

PHYS_OFFSET

Physical start address of the first bank of RAM.

PAGE_OFFSET

Virtual start address of the first bank of RAM. During the kernel boot phase, virtual address PAGE_OFFSET will be mapped to physical address PHYS_OFFSET, along with any other mappings you supply. This should be the same value as TASK_SIZE.

TASK_SIZE

The maximum size of a user process in bytes. Since user space always starts at zero, this is the maximum address that a user process can access+1. The user space stack grows down from this address.

Any virtual address below TASK_SIZE is deemed to be a user process area, and therefore managed dynamically on a process by process basis by the kernel. This is referred to as the 'user segment'.

Anything above TASK_SIZE is common to all processes. This is referred to as the 'kernel segment'.

Note: This means that you can't put IO mappings below TASK_SIZE, and hence PAGE_OFFSET.

TEXTADDR

Virtual start address of kernel, normally PAGE_OFFSET + 0x8000. This is where the kernel image ends up. With the latest kernels, it must be located at

32768 bytes into a 128MB region. Previous kernels just required it to be in the first 256MB region.

DATAADDR

Virtual address for the kernel data segment. Must not be defined when using the decompressor.

VMALLOC_START, VMALLOC_END

Virtual addresses bounding the vmalloc() area. There must not be any static mappings in this area; vmalloc will overwrite them. The addresses must also be in the kernel segment (see above). Normally, the vmalloc() area starts VMALLOC_OFFSET bytes above the last virtual RAM address (found using variable high_memory).

VMALLOC_OFFSET

Offset normally incorporated into VMALLOC_START to provide a hole between virtual RAM and the vmalloc area. We do this to allow out of bounds memory accesses (e.g., something writing off the end of the mapped memory map) to be caught. Normally set to 8MB.

9.6.3. Architecture Specific Macros

BOOT_MEM(pram,pio,vio)

'pram' specifies the physical start address of RAM. Must always be present, and should be the same as PHYS_OFFSET.

'pio' is the physical address of an 8MB region containing IO for use with the debugging macros in arch/arm/kernel/debug-armv.S.

'vio' is the virtual address of the 8MB debugging region.

It is expected that the debugging region will be re-initialised by the architecture specific code later in the code (via the MAPIO function).

BOOT_PARAMS

Same as, and see PARAMS_PHYS.

FIXUP(func)

Machine specific fixups, run before memory subsystems have been initialised.

MAPIO(func)

Machine specific function to map IO areas (including the debug region above).

INITIRQ(func)

Machine specific function to initialise interrupts.

9.7. Kernel Porting

Finally we get to the meat of the task. Here we list the most important files, and describe their purpose and the sort of things you should put in them. It looks daunting to start with but most of what is required is just a matter of filling in the numbers appropriate to your hardware. Now that so many different machines are supported it is rare that you have to write much new code - nearly everything can be taken from a suitable donor machine. This is easier to do if you know which machines have a similar architecture to your own.

Throughout this list XXX represents your machine name - 'anakin' in these examples

Files

`arch/arm/Makefile`

Insert the following to this file (replace **XXX** with your machine name):

```
ifeq ((CONFIG_ARCH_XXX),y)
MACHINE                 = xxx
endif
```

`arch/arm/boot/Makefile`

you specify **ZTEXTADDR**, the start address of the kernel decompressor. This should have the same value as the *address to which Linux is uploaded* in your boot loader. This is normally 32K (0x8000) above the base of RAM. The space between the start of RAM and the kernel is used for page tables.

```
ifeq ((CONFIG_ARCH_XXX),y)
ZTEXTADDR       = 0xXXXX8000
endif
```

`arch/arm/kernel/entry-armv.S`

Machine-specific IRQ functions. You provide the assembly macros disable_fiq, get_irqnr_and_base, and irq_prio_table here. disable_fiq and

irq_prio_table is usually empty, but get_irqnr_and_base must be implemented carefully: you should use the zero flag to indicate the presence of interrupts, and put the correct IRQ number in irqnr.

`arch/arm/kernel/debug-armv.S`

These are the low-level debug functions, which talk to a serial port without relying on interrupts or any other kernel functionality. You'll need to use these functions if it won't boot. They are included in the kernel when `CONFIG_DEBUG_LL` (Low Level Debugging) is defined. The functions you need to implement are `addruart`, `senduart` and `waituart`, using ARM assembly. They give you the address of the debug UART, send a byte to the debug UART, and wait for the debug UART, respectively. You normally use these by calling the `printascii()` function.

`arch/arm/mach-XXX/Makefile`

You need to add a target for your machine, listing the object files in this directory. That will be at least the following:

```
+O_TARGET               := MACHINE.o
+obj-y                  := arch.o irq.o mm.o
```

`arch/arm/mach-XXX/arch.c`

This should contain the architecture-specific fix ups and IO initialisations. (The latter used to go in `arch/arm/mach-XXX/mm.c` but that file is now deprecated). For the meaning of the symbols and macros, please refer to Section 9.6.

The setup for your machine is done with a set of macros, starting with `MACHINE_START`. The parameters you give are filled in to a data structure machine_desc describing the machine. One of the items is the `fixup_XXX` function which, if specified, will be called to fill in or adjust entries dynamically at boot time. This is useful for detecting optional items needed at boot time (e.g. VRAM in a Risc PC).

> **Note:** It should not be used to do main memory size detection, which is is the job of the boot loader.

For the IO initialisation you fill in a struct map_desc and pass it to `iotable_init`.

```
static void  __init
fixup_XXX(struct machine_desc *desc, struct param_struct *params,
             char **cmdline, struct meminfo *mi)
```

```
{
       ROOT_DEV = MKDEV(RAMDISK_MAJOR, 0);
       setup_ramdisk(1, 0, 0, CONFIG_BLK_DEV_RAM_SIZE);
       setup_initrd(0xc0800000, X * 1024 * 1024);
}

MACHINE_START(ANAKIN, "XXX")
       MAINTAINER("Acunia N.V.")
       BOOT_MEM(XXX, YYY, ZZZ)
       VIDEO(VVV, WWW)
       FIXUP(fixup_XXX)
       MAPIO(XXX_map_io)
       INITIRQ(genarch_init_irq)
MACHINE_END

static struct map_desc XXX_io_desc[]   __initdata = {
     {    IO_BASE,    IO_START,    IO_SIZE,    DOMAIN_IO, 0, 1, 0, 0 }
        LAST_DESC
};

void   __init
XXX_map_io(void)
{
       iotable_init(XXX_io_desc);
}
```

`arch/arm/mach-XXX/irq.c`

You should provide the `XXX_init_irq` function here. This sets up the interrupt controller. Interrupt mask and unmask functions go here too.

`arch/arm/mach-XXX/mm.c`

This file is now deprecated. Its content (IO initialisation) has moved into `arch/arm/mach-XXX/arch.c`

include/asm-arm/arch-XXX/dma.h

Defines for DMA channels, and DMA-able areas of memory. For machines without DMA, you can just declare 0 DMA channels as follows:

```
#define MAX_DMA_ADDRESS         0xffffffff
#define MAX_DMA_CHANNELS        0
```

`include/asm-arm/arch-XXX/hardware.h`

In this file, you need to define the memory addresses, IO addresses, and so on, according to your hardware specifications (memory map and IO map).The _START addresses are the physical addresses, the _BASE addresses are the virtual adresses to which each memory or IO region will be mapped. Refer to other similar machines for examples.

`include/asm-arm/arch-XXX/io.h`

Please define the macros IO_SPACE_LIMIT (as 0xffffffff), __io(addr), __arch_getw(addr), __arch_putw(data, addr), and other related macros, according to your CPU. For CPUs that already have an implementation (for example SA1110 and XScale), you can just copy it across and/or reuse the existing `io.h` file for that CPU.

`include/asm-arm/arch-XXX/irq.h`

Here you need to provide the fixup_irq macro. In almost all cases, this is just a direct mapping:

```
#define fixup_irq(i)            i
```

`include/asm-arm/arch-XXX/irqs.h`

In this file you will define all your IRQ numbers. For example:

```
#define IRQ_UART0               0
```

`include/asm-arm/arch-XXX/keyboard.h`

This file is typically here to cheat the VT driver into thinking that there is a null keyboard. Most ARM devices don’t have a real one. If yours does this is where to put the keyboard IO defines and structs.

`include/asm-arm/arch-XXX/memory.h`

Unless you have an exotic memory-map this is platform-invariant and you can copy this from other implementations.

`include/asm-arm/arch-XXX/param.h`

This is included by `asm/param.h`. Here you can redefine HZ (default 100), NGROUPS (default -1), and MAXHOSTNAMELEN (default 64). If you are okay with the above defaults, you still need to create this file but you can make it an empty file (as it is in the Anakin case).

`include/asm-arm/arch-XXX/system.h`

This file is included by `arch/arm/kernel/process.c`. You are required to

define `arch_idle()` and `arch_reset()` functions. `arch_idle()` is called whenever the machine has no other process to run - i.e. it's time to sleep. arch_reset() is called on system reset. The actual implementation of these functions depends on your specific hardware, and there are some subtleties associated with `arch_idle()`. This function will normally put the hardware of your specific device into a low-power mode and then call the generic CPU function `cpu_do_idle` to do the same thing for the CPU. A typical implementation would be as in the listing below, however in Anakin's case this won't work, because the interrupt controller is in the ASIC, and that is clocked by the processor's mclk. Stopping the CPU stops the ASIC as well, which means that a wake-up interrupt will never get generated, so calling `cpu_do_idle` just hangs forever. So for Anakin, the `arch_idle` function does nothing. You need to know about this sort of hardware detail to get a successful port.

```
static inline void arch_idle(void)
{
        /*
         * Please check include/asm/proc-fns.h, include/asm/cpu-*.h
         * and arch/arm/mm/proc-*.S.  In particular, cpu_do_idle is
         * a macro expanding into cpu_XXX_do_idle, where XXX is the
         * CPU configuration, e.g. arm920, sa110, xs80200, etc.
         */
        cpu_do_idle(IDLE_WAIT_SLOW);
}

static inline void arch_reset(char mode)
{
        switch (mode) {
        case 's':
                /* Software reset (jump to address 0) */
                cpu_reset(0);
                break;
        case 'h':
                /* TODO: hardware reset */
        }
}
```

`include/asm-arm/arch-XXX/time.h`

Here you have to supply your timer interrupt handler and related functions. See the template below:

```
/*
 * XXX_gettimeoffset is not mandatory.  For example, anakin has
```

```
 * not yet implemented it.  dummy_gettimeoffset (defined in
 * arch/arm/kernel/time.c) is the default handler, if you omit
 * XXX_gettimeoffset.
 */
static unsigned long XXX_gettimeoffset(void)
{
        /* Return number of microseconds since last interrupt */
}

static void XXX_timer_interrupt(int irq, void *dev_id, struct
        pt_regs *regs)
{
        /* Add hardware specific stuffs, if applicable */
        do_timer(regs);
        do_profile(regs);
}

extern inline void setup_timer(void)
{
       gettimeoffset = XXX_gettimeoffset;
       timer_irq.handler = XXX_timer_interrupt;
       setup_arm_irq(IRQ_XXX, &timer_irq);
       /* Other hardware specific setups */
}
```

The `do_profile()` function is there to allow kernel profiling.

`include/asm-arm/arch-XXX/timex.h`

This file is included by `include/asm/timex.h`, which is in turn included by `include/linux/timex.h`. Basically you need to define your clock rate here. For example, for Anakin it is *1 / 8ms*:

```
#define CLOCK_TICK_RATE                 1000 / 8
```

`include/asm-arm/arch-XXX/uncompress.h`

This file is included by `arch/arm/boot/compressed/misc.c` (which, among other things, outputs the `Uncompressing Linux` message). You are required to provide two functions, `arch_decomp_setup` to setup the UART, and `puts` for outputting the decompression message to the UART:

```
static void_puts(char char *s)

        /*
         * Hardware-specific routine to put a string to the debug
         * UART, converting "\n" to "\n\r" on the way.
```

```
                */


        static inline void arch_decomp_setup(void)

                /*
                 * Hardware-specific routine to put a string to setup the
                 * UART mentioned above.
                 */


        /* Watchdog is not used for most ARM Linux implementations */
        #define arch_decomp_wdog()
```

`include/asm-arm/arch-XXX/vmalloc.h`

This file is largely invariant across platforms, so you can just copy it from other ARM architectures without worrying too much. Make sure that the mappings in here and the static mappings in `include/asm/arch-XXX/hardware.h` do not overlap.

If you get all the above right then you should have a bootable compressed kernel for your architecture that can output debug messages through the debug functions. However, this isn't actually much use without some support for the devices in your system. In Anakin's case this is the frame buffer, UARTs under interrupt control, touchscreen and compact flash interface (IDE). Console driver functionality is also required to actually interact with these devices. We'll look at how to add these drivers in a future chapter.

9.8. Further Information

A number of resources are also available both on-line and in book format.

The ARM Linux Project

http://www.arm.linux.org.uk/

Linux Console Project

http://sourceforge.net/projects/linuxconsole/

Linux Framebuffer Driver Writing HOWTO

http://www.linux-fbdev.org/HOWTO/

9.9. Background

The first edition of our book *A Guide to ARM Linux for Developers* is available now and the second edition from which this chapter is taken will be out later this year. Please refer to http://www.aleph1.co.uk/armlinux/thebook.html for further information.

Thanks to Russell King for his help and input on this chapter.

Chapter 10. Linux Overview

This chapter describes some GNU/Linux fundamentals and tries to teach by example some basic things you need to know in order to use a GNU/Linux system, e.g. how to edit a configuration file, or print something. It also teaches you about some useful commands and concepts. It is written assuming that you are running an ARMLinux desktop system, but largely applies equally to any desktop system. If you have not used GNU/Linux much or at all before then this chapter will be very useful. Although written from a desktop perspective, some of the info here is relevant to small ARMLinux systems too.

10.1. Logging in

After installing GNU/Linux, either on the desktop, or on a target device and booting up, you should be met by the following:

armlinux login:

If this is the first time that you have logged in, then you will have to initially log in as the root user which is accomplished by typing **root** before pressing **Return**.

You will then see that the prompt is represented by a hash sign (#) which means that you are looking at the *root* prompt. In contrast, the prompt for non-root users is usually represented by a dollar sign.

[root@armlinux /root]#

or for the non-root user:

[paulw@armlinux paulw]$

Although it is possible to work with your new Linux system at this stage, we would advise you to immediately refer to Section 10.6 for the specifics of creating a user account.

10.2. The Shell

As a new ARMLinux user, you will have to get to grips with the Unix command prompt although memory-efficient GUIs like fvwm are available. Regular users of

Linux call this prompt a *shell* and it is the means by which you can communicate with the kernel. In fact, the shell is a program which you can use to do all your Linux work if this is what you desire. Because Linux has been developed by a range of people with different interests and needs, it is not surprising that there are a range of shells that you can choose from. Examples include the Bourne Again Shell (Bash) and the Korn Shell (ksh). To find out the name of your current shell type the following command at the prompt:

echo $SHELL

If the response is: `/bin/bash` then your default shell is bash. Bash is a good shell to learn both because it is installed by default on many Linux distributions and because it has a range of features which you will learn to use as your Linux experience increases. It can also be a good idea to learn how to work with a second shell in case you ever have to use a machine that is not running Bash.

10.3. Simple Commands

There are also a number of simple commands that you can employ to do useful work on your Linux system like those which are shown in Table 10-1.

Although we list a range of options which you can use with these commands, you should learn how to refer to the Unix *manual* pages for a definitive explanation (see Section 10.5).

Table 10-1. Simple Commands

Command	Explanation
cd	change directory
pwd	print working directory
ls	list files
mv	move a file
cp	copy a file
rm	delete a file
mkdir	make a directory
rmdir	delete a directory

10.3.1. Change Directory

The **cd** command allows you to *change directory*. To change to the `/home` directory, you would therefore type **cd** and `/home` at the command line.

cd /home

10.3.2. Print the Working Directory

You can also find out the location of the current directory as an *absolute path* from root by using the **pwd** command to display the result on screen. Our example consequently shows that we are currently in the `/home` directory.

pwd
`/home`

10.3.3. List Files

The *list files* command is invoked by typing **ls** in combination with a range of arguments which are shown in Table 10-2.

Table 10-2. The List Command

ls Command with Options	Explanation
ls -a	list all files
ls -l	long-format (detailed) listing
ls --color	list filetypes in colour
ls -F	show filetype
ls -i	produce a detailed listing
ls -s	print file size

Tip: In order to take advantage of the *listing filetype by colour* facility which **ls** offers, you may need to establish an *alias*. Assuming that you are using Bash, enter the following line in your `.bashrc` file.

```
ls='ls --color'
```

10.3.4. Move Files

The **mv** command is used to move a file or directory from one place to another. The file `sample.txt` can therefore be moved into your `/home` directory by issuing the following command where the tilde (~) represents `/home`.

mv sample.txt ~

mv can also be used to rename a file. To change `sample.txt`'s name to `final.txt` and move the renamed file to the `/home` directory, enter the following command:

mv sample.txt ~/final.txt

10.3.5. Copy a File

cp is very similar to **mv** in terms of the way in which you use it. One notable difference is that **cp** leaves the source file in situ whilst *copying* the file to your intended destination.

cp sample.txt ~/home

> **Tip: cp** is also used to copy directories including their contents. If you need to do this, issue the **cp** command in conjunction with the `-R` option before typing the names of your source and destination directories.

10.3.6. Delete a File

The **rm** command is very easy to use although it is extremely easy to delete files unintentionally! We therefore advise you to use the `-i` option so that Linux confirms your intentions before taking any action.

rm `-i sample.txt`

10.3.7. Make a directory

You can use the **mkdir** command to create directories and sub-directories. Creation of a directory therefore involves typing the command and the name of your proposed directory whereas the creation of a sub-directory depends upon the `-p` option.

mkdir `work`

mkdir `-p /work/paul`

10.3.8. Delete a Directory

It's easy to delete an empty directory with the **rmdir** command. If we assume that the `pleasure` directory is empty, the next command removes it from our system.

rmdir `pleasure`

> **Tip:** You should use the **rm** with the `-R` option to delete a directory which contains files.

10.4. The Filesystem

In order to provide you with a working knowledge of the Linux filesystem, the following topics will be introduced in this section.

1. The Linux Root Filesystem
2. Permissions
3. Alternative Filesystems

10.4.1. Overview

The filesystem is made up of *files* and *folders* or directories. The top level folder is called the *root* folder and contains all of the files that are pertinent to your system. Each file within this system is allotted a name and a path on the basis of where it lies in relation to the root folder. Thus a sample file -`alephlinux.txt`- may be found at `/home/letters/alephlinux.txt` where the forward slash "/" before *home* denotes root.

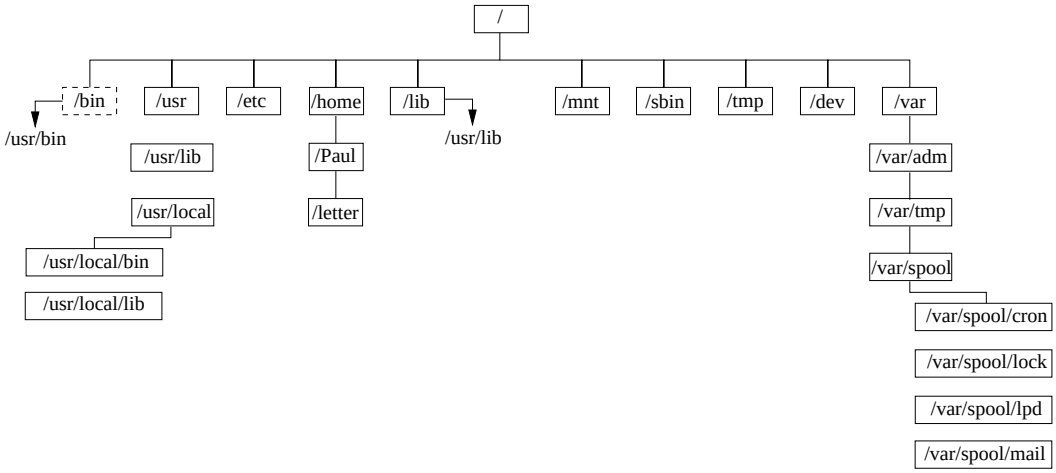
> **Important:** The new user must also appreciate that the filesystem is case-sensitive and therefore distinguishes between `ALEPHLINUX.txt` and `alephlinux.txt`.

Linux also provides support for long filenames although you should avoid the use of symbols like "@" within filenames, as they are used for other purposes.

10.4.2. The Linux Root Filesystem

The following schematic provides a simple representation of the most important directories and files which make up the root filesystem whilst the subsequent sub-section provides a brief description of the filesystem's structure. For more information about the root filesystem, enter **man hier** at the prompt for a full description.

Figure 10-1. Linux Root Filesystem



Linux-Root Filesystem

Directories

`/usr`

Contains files and sub-directories which may be shared between machines. Important sub-directories include:

- `/usr/lib` which contains programming and language libraries
- `/usr/local` which provides the location for user-specific programs and files
- `/usr/include` for C #include files

`/etc`

all configuration files should be in here (often in subdirectories)

`/home`

contains home directories for all users on the system. `/home/paul` would therefore be the home directory for user *paul*.

`/bin`

contains fundamental executable files (commands or programs) used by users. Only the basic ones are here, most are in /usr/local/bin, but usually a user doesn't care where the executable is actually stored. **which <commandname>** will tell you where an executable actually resides.

`/sbin`

includes executable files which are for system administration, as opposed to programs used by users. An example of this would be the **shutdown** command which closes the system down properly.

`/mnt`

location for *mounting* filesystems. For supplementary information, consult Section 10.4.4

`/tmp`

a directory which provides the temporary location for different programs. `/tmp` files tend to be deleted when the system is re-booted.

`/dev`

contains special *pseudo* files which give access to devices like the keyboard - /dev/kbd - and the virtual terminals - /dev/console, /dev/tty0 and /dev/tty1.

`/var`

a directory which contains files which can grow through usage. Typical examples include email message files and system accounting files.

- `/var/adm` for per-process accounting records
- `/var/tmp` for temporary files which are not deleted at re-boot time.
- `/var/spool` for running system processes. /var/spool/mail would be a typical file in this sub-directory

10.4.3. Permissions

Although you should now have some appreciation of the Linux filesystem it is important to realise that Linux can be used in a multi-user or networked environment. It is therefore sensible to learn about the various types of *role* and *permission* which can be associated with a file so that the file *owner* can decide exactly who has access to files.

Any user may consequently fit into any one of three roles which include:

1. the *ownership* role
2. the *group* role
3. the *other* role

The *owner* of a file can change file permissions at will. Moreover, a file may be associated with a *group* which is useful in a collaborative project context in which file access can be restricted to group members. Finally, the *other* role applies to a user who is neither the owner of the file nor the member of the group which is associated with the file.

In a similar vein, for every Linux file, three types of permission can be applied to each of the three roles. These permission *switches* are as follows

1. R (Read): enables the user to read files, or (for a directory) to list files on the system
2. W (Write): allows for the alteration or deletion of files, or (for a directory) deleting, adding and renameing of files
3. Execute (Execute): allows for the execution of a file where it is an executable or for changing directories.

In order to clarify matters, this information can be most effectively presented in tabular form. Note that a dash in the line of text denotes the absence of a permission whilst r, w or e indicates that one of the three types of switch is activated.

Table 10-3. Permissions by Role

Owner	Group	Other
r w x	r w x	r - x

To illustrate all of the above, consider a situation where we want to know the permissions which apply to a particular file. After invoking the **ls -l** <filename> command, our hypothetical output might look some thing like the following:

—rw—r——r—— 1 paul paul 388 June 10 09:30 notes.txt

Reading across the listing from left to right, the first character position show the file's filetype which in this example is a normal file. Other possibilities include:

1. d for a directory
2. c for a device
3. b for a socket
4. s for a pipe

The remaining nine bits apply to the user, group and other roles respectively with three bits being alotted to each of the roles. Thus our hypothetical output shows that the user has *read* and *write* permissions whilst the *group* and *other* roles have read permission only. Note also that a dash denotes the absence of a permission.

If however, it becomes necessary to change file permissions, this can be accomplished by:

1. setting file permissions with octal numbers
2. setting file permissions with a combination of symbols
3. setting the setuid, setgid and sticky bits

To alter a file's permissions with numbers, Linux uses four numbers which determine permission types including:

1. 4: read permissions
2. 2: write permissions
3. 1: execute permissions
4. 0: no permissions

Therefore, to alter the permissions status of our hypothetical file so that the *group* has *read* and *write* permissions, we would add 4 (read permission) and 2 (write permission) together before issuing the following command:

chmod 664 notes.txt

The following table summarises the situation thus far:

Table 10-4. Permissions by Numbers

Role	Permission	Calculation
Owner	6	4+2+0 (r+w+e)
Group	6	4+2+0 (r+w)
Other	4	4+0+0 (r)

In a situation where it is necessary to set permissions one role and one permission at a time, the *symbolic permission* notation can be used. In order to make use of this facility, the user need only remember that permissions may be granted or denied by using the following three symbols in conjunction with the read, write and execute switches that we have already encountered:

1. u signifies the user/owner role
2. g represents the group role
3. o signifies the other role

Therefore, in order to allocate a group read and write permissions for the file `notes.txt` using symbolic notation, the user could enter the following command:

chmod g+rw notes.txt

Because many permutations are possible when using this method, the table below gives some indication of how permissions may be alterred. Note that "+" means that *permission is granted* whilst "-" shows that *permission is denied*.

Table 10-5. Permissions by Symbols

Granted/Denied	Permissions	String
user, group, other	read, write, execute	ugo+rwx
user, group, other	read, write, execute	ugo-rwx
user, group	read, write	ug+rw
user, group	read, write	ug-rw
user	read	u+r
user	read	u-r

Of course, there is much more that could be said about permissions but for further details on the specifics of topics like sticky bits, shadow passwords and filesystem maintenance, consult the relevant **man** page. Refer to Section 10.5 for advice on using **man**.

The important thing to remember is that if you are having a problem with *missing files* or some other mysterious error, it is very often due to incorrect permissions somewhere in the system. Always remember to check.

10.4.4. Alternative Filesystems

ARMLinux can similarly make use of a range of filesystems in addition to its own Second Extended filesystem (ext2fs).

Under Linux the user will however have to *mount a device* before it can be accessed

and used. This is done with the **mount** command but you will initially have to edit the `/etc/fstab` file as the root user in order to avail of this facility. This file is also used to specify which filesystems should be mounted automatically at boot time. To edit this file - which is just a text file - use Vim to open the file with the following command:

vim /etc/fstab

If your file doesn't include a line like the following:

```
/dev/hda1    /adfs    adfs      noauto,user   0   0
```

then you should add one in order to be able to read your RISC OS drive from ARMLinux. Note that if your RISC OS drive is RISC OS 4 with long filenames then this is not supported at the time of writing so you can skip this section. The format of this file is explained in **man fstab**.Save and quit by depressing the **ESC** key, waiting for a beep and then typing **:wq /etc/fstab**. Where this line is missing, type it in from scratch noting that there should be no spaces between *noauto* any punctuation and *user*. if you want this drive to be automatically mounted on boot then change the final 0 to a 1. You will then need to make a mount directory.

mkdir /adfs

Congratulations, you are now in a position to mount your RISC OS filesystem with the following command:

mount /adfs

You can now access your files from the `/adfs` directory.

10.4.5. Debian Package Management

In order to take full advantage of Debian, you need to learn a little about the package management system which underpins it. Debian packages everything in .DEB files. These packages are controlled by a program called dpkg, or from a friendlier front-end such as dselect or apt. The system is extremely powerful; every file on your system is known about by the system and it takes care of a great deal of complexity when installing, removing or upgrading software. It knows about the interdependencies between pieces of software and if you ask to install something it will also get the other things it needs. It knows which files are configuration files and will not overwrite them when upgrading a package. It can also upgrade a running system without breaking things and lets you find out what is on your system. To a large extent the package management system is what makes Debian special and once you understand it you will find it a marvellous facility.

dselect is the interface that you will first use. It was designed when Debian had 500 rather than 5000 packages and suffers a bit because of this, but it is quite straightforward to use and a good introduction to the package system. Later you will just use apt to install new software. Once apt has been taught where to get software (usually off the CD and off the net) you can say things like **apt-get install <software>** and it will go away and get it for you, then install it, install anything else necessary and configure them all, often without asking you any further questions apart from telling you which CD to insert or if the download size is OK.

It is important to understand however that if you want to install software that is not in the Debian archive then everything is not taken care of for you and you will have to understand what is going on and do some work yourself. For example, if obtaining a package from a distribution which is meant for x86-based PCs, you will have to compile it from source as the binary versions supplied will not work on an ARM-based machine. Such a procedure is outside this manual's scope but - assuming that you want to use a package which is on the ARMLinux CD or in the huge Debian archive - read on!

dpkg can for example be used to install a package by using the `i` (or --install) switch.

dpkg -i /cdrom/debian/dists/potato/main/binary-arm/utils/mc<TAB>

Note that the <TAB> above means press the TAB key, not type in those letters. This uses the shell's *filename expansion* facility to fill in the end of the filename. It's a marvellous facility. If it beeps then there is more than one possible answer - press <TAB> again to see the list. It even works on commands.

However, it is standard practice to use apt instead to save typing long pathnames, like this:

apt-get install mc

Here apt knows where to look for packages, finds mc and then uses dpkg to actually do the installation, but saves you worrying about the actual location of the package.

What you have just done is to use dpkg to install Midnight Commander (mc) - a visual shell which does not need X Windows - on your system. Conversely, you may choose to delete or uninstall a package which can be accomplished by using the `-r` (or --remove) switch. The following command consequently removes mc.

dpkg -r mc

Don't be too hasty however. MC is an excellent way of navigating the filesystem when used in conjunction with the shell. So why not learn a bit about it before using dpkg's `-r` switch?

Congratulations, you've managed to mount your filesystem and have a look at some files. Before removing your CD-ROM, you will need to use the **umount** command which can either be used to unmount the current filesystem or a specific directory.

But please be careful with your typing and key in **umount** rather than **unmount**. If it tells you that the `filesystem is busy` then your current directory is probably in the mounted filesystem. You need to change directory elsewhere before you can **umount**.

umount /dev/cdrom

or

umount /cdrom

In order to find out about other filesystem types and the specifics of the **mount** and **umount** commands, you can take advantage of the range of commands and applications which are briefly described in Section 10.5.

10.5. Help!

If you are an experienced Linux user, enter the following commands before proceeding to the next section. If you are a beginner, move swiftly on to the next paragraph!

updatedb

/usr/sbin/makewhatis

If you are a newbie, you will be a little perplexed by all the unfamiliar terminology. If this is the case, don't despair. Linux is *very* well documented and you can access all this information by taking a few preliminary steps. At the prompt enter the following command:

updatedb

This command updates a database of filenames which you can then search by typing in the **locate** command with your filename as the command's argument. The experienced user can of course enter special characters - called *metacharacters* - as opposed to a text string in order to generate a more sophisticated search but a discussion of this technique is outside the scope of this manual. For further details take a look at the **updatedb** man page as well as the man pages for **locatedb**, **locate**, **find** and **xargs**. Remember however to update the filename database regularly if you are continually producing new documents. If your machine is left on overnight it will automatically do this itself.

You can of course make use of a further command which you can again enter at the shell prompt (and as root):

/usr/sbin/makewhatis

What you have just done is to build the *whatis* database which contains information that you can now access to improve your skills. This process may take a few minutes but when it is completed, you can read the Unix manual pages on any subject that you want. To make use of man, type the **man** command with the topic area as man's argument at the shell prompt. You can then scroll through the manual by pressing the space bar as a prelude to leaving man by typing q for *quit*. The following command consequently takes you to that part of the Unix manual which is concerned with Groff - a document formatting system which many people use as an alternative to a word processor:

man groff

But how can you use man if you don't know the name of the command which would meet your requirements? Well, help is available with the **apropos** command. To use **apropos**, simply type the name of a keyword that is applicable to your interests after the apropos command (it is this indexing ability that the *whatis* database is used for). In order to find out about the editors which are available on your system type:

apropos editor

You may also see **man -k** mentioned - which is exactly equivalent to **apropos** (use whichever command you prefer) - as well as the **whatis** command. **Whatis** is particularly useful if you have heard about an application and require a simple one line description. If necessary, you may also take advantage of the built-in help facility which comes with many commands by typing the name of a command, a help switch, the *pipe* character and **less** to signify the less pager which allows you to view information one screen at a time.

man --help | less

Alternatives to both **man** and the **apropos** command include the info pages, the manuals of the Linux Documentation Project which can be downloaded from (http://www.linuxdoc.org/) and a selection of HOWTO documents which are available on the Aleph ARMLinux CD and can be read through a web browser like Lynx.

10.6. Accounts

You may also need to create a *user account* if you have not done this already. It is important to do this because the *root* account gives you access to *all* files on your system. What this means is that a few careless strokes on the keyboard whilst you are logged in as root could result in an unusable system (**rm** is a potentially disastrous command to use although its effect can be lessened with the `-i` switch). In your early adventures with ARMLinux it would therefore be better if you learnt

about your system from the relative safety of a user account. To create a user account, follow the instructions below:

Login as root,
or
su root
if logged in as an ordinary user,
type
adduser fred
at the prompt where *fred* is our hypothetical user,
You will be asked to set a password, (which can be done for an existing user with the **passwd** command)
and follow the instructions.

Assuming that you are logged in as root do the following:

adduser majcon

Linux will set up the new user with a password, user id, a group membership (id) and a login name, home directory and mailbox. You can then enter the password twice as instructed.

A detailed discussion of Linux systems administration is outside the scope of this manual but consult *The Linux Systems Administrators' Manual* which is available from http://www.tcm.hut.fi/~viu/linux/sag/ for more information. What you must remember is that your password must be as ingenious as you can make it if you wish to avoid the possibility of your files being read by a stranger. Your password should make use of a mixture of upper and lower-case letters as well as numbers and symbols. But again, be sure to remember what you have actually typed!

You might think that this discussion of user accounts is not really relevant to you but it becomes an important consideration when your ARMLinux machine is being used concurrently by other users in a business or educational setting.

10.7. Editing

Because of the many uses to which ARMLinux can be put, it would not be practical in a short manual to provide a description of the rudiments of every program. We have therefore limited ourselves to an introductory treatment of text editing because this is a task which any user will need in their ARMLinux career.

There are literally hundreds of editors available including Vim, Jed, Joe and Emacs. Jed is a nice full-screen editor which is somewhere between MS DOS's Edit and Emacs whilst Joe will be appreciated by those who are used to the old

Borland/WordPerfect *control-key* settings. Emacs, in contrast, has many of the characteristics of a complete working environment because it allows the user to do everything from writing letters to surfing the net. You can even play games within Emacs or indulge in a bit of psychotherapy by using the *doctor* package! We have not however, provided an Emacs tutorial in this guide so interested readers should refer to Section 13.8 or to http://www.gnu.org/manual/emacs-20.3/emacs.html/ where you can find a comprehensive Emacs guide.

> **Tip:** If you decide to learn Emacs, you may see references to the META key in the Emacs literature which is equivalent to the ALT key on most keyboards. If you can't use ALT (it often doesn't work in terminals, for example) then you can use ESC to do the same job, but it should be released before the next keypress, not held down at the same time as it.

This section introduces you to:

1. ae - the standard, tiny editor.
2. Jed - a *modeless* editor which users of StrongED and Zap will find very accessible.
3. Vim - an extremely powerful editor which can be found on all Unix-like operating systems.

10.7.1. Editing with ae

ae can be activated by typing:

1. **ae** at the shell prompt
2. **ae <filename>** where you wish to work on a pre-existing file.

Two screens will then appear: an *editing* screen into which you can type and a *help* screen which provides a selection of useful commands. If you have used ae before, you can toggle the help screen *on* and *off* by typing C-x+C-h.

ae operates, like many Linux editors, via a series of *control characters*. C-x therefore instructs the user to depress the **C** (**Ctrl**) and **x** keys at the same time.

Typing a *non-control character* consequently has the the effect of placing the character which you have typed at the cursor. When you have finished your work, type C-x+C-s to save the current file to disc or C-x+C-c to save and exit.

ae is therefore very easy to use although its simplicity conceals a number of sophisticated features which include:

- The ability to work with multiple buffers by using a root buffer and a series of virtual folds which correspond to the files to be edited.
- User configurable key bindings.

For further information on ae, visit the editor's home page at http://www.interalpha.net/customer/nyangau/ae/platform.htm/.

10.7.2. Editing with Jed

Jed can be driven from a series of menus or with a combination of shortcuts. Jed is a *modeless* editor which means that you do not have to be in different modes to *edit* or *insert* text. Jed is also very easy to use and offers a range of facilities which include:

- Colour syntax highlighting which can be useful for debugging your documents.
- Special features which are activated on the basis of filetype.
- Extensibility via the C-like S language.

Jed can be invoked by typing:

1. **jed** at the shell prompt.
2. **jed <filename>** at the shell prompt where you wish to edit a specific file.
3. C-x+C-f in the mini-buffer. If you choose this command, you will see `Enter Filename` in the mini-buffer.

In addition to commands which begin with the **Ctrl** character, Jed also uses the **ESC** key for a number of commands. **ESCn** therefore means that you should press **ESC** before *releasing* it and pressing key **n**.

Assuming that you are working on the file `readme.txt`, type **jed <readme.txt>** at the shell prompt. You can then insert and edit text and move around the document by using the *arrow* keys. When you have finished the session, type C-x+C-s to save your document to disc. You can then exit from Jed by typing C-x+C-c.

The *Jed screen* contains a number of features which may be unfamiliar to you. These are:

1. A Menu Bar across the top of the screen.
2. A Mode Line containing useful information.
3. A Mini-Buffer which is below the Mode Line.

The *Menu Bar* contains the menu options which you can use as an alternative to the keyboard shortcuts. To activate the Menu Bar, press the **F10** key and select the menu which you require by using the arrow keys. The menu options should be

self-explanatory with the possible exception of the Buffers option. A buffer is essentially a temporary workspace inside which you perform your work until you save any changes which are then written to the file.

In contrast, the *Mode Line* provides information about the document which you are currently working on. If you can see two asterisks (**) on the extreme left of the mode line, this is an indication that you have made changes to a document which has not been saved to disc. The mode line also tells you where you are in the document in terms of a line number as well as giving an indication of the current *editing mode*. Editing modes essentially help you to efficiently edit your documents by activating features like syntax colouring and specific formatting styles.

Finally, the *Mini-Buffer* is located below the Mode Line and *echoes* commands back to the user. Try keying in C-x+C-v to *find an alternative file* and you should get the idea but remember to hit the **Return** button after you have finished typing.

You should now know enough about Jed to perform simple editing. Jed also comes with its own form of online Help which can be activated by pressing C-h+C-h. A number of useful WWW resources are also available so point your browser to http://space.mit.edu/%7Edavis/doc/jed.html/ and http://www.geocities.com/mmarko.geo/jed/ Finally, take a look at Appendix A which lists some useful Jed commands. Please be aware however that the table uses commands which assume that Jed is using Emacs keybindings.

10.7.3. Editing with Vim

Vim is another editor which you can learn how to use. Vim also comes with an excellent tutorial which you can avail of by typing the following command at the prompt:

vimtutor

You can also save a bit of typing by using the *filename completion* facilities of the Bash shell. Simply type **vim/usr/doc/vim** and then press <TAB> so the shell fills in the remainder of the directory name, then you can enter the `tutor` and press <TAB> again to get the rest.

The tutorial will open from within Vim and you will then be able to learn enough to perform most of your editing activities. But just in case you want to familiarise yourself with Vim without recourse to the tutorial, what follows is an introduction to the editor’s capabilities.

Vim is a *modal* editor which means that there are different modes for performing different tasks. Vim consequently operates in *command* mode when commands are being issued or in *insert* mode when text is entered on the screen. Although the method of issuing commands with a key press may seem counter-intuitive and

positively archaic in comparison with more conventional point-and-click methods, it is actually a very efficient method of working and should be perservered with if at all possible.

In order to become conversant with Vim, we therefore suggest that you learn a few new commands per day and practice! Appendix B lists some of the common commands that you are likely to need.

> **Note:** The user enters command mode by pressing the **ESC** key whilst insert mode can be activated by pressing i.

For more information on Vim, consult its home page http://www.vim.org/ or check out one of the many books which are available on Vi - Vim's predecessor.

10.8. Linux and Printing

10.8.1. Overview

After having learnt something about text editing, it seems sensible to provide a brief introduction to the subject of printing under Linux. Although word processors are available for Linux, you will still need to learn something about traditional Unix tools like the **lpr** printer daemon and the power of PostScript - a language which is used to describe page layouts.

Although using the command line to print documents is not the easiest way to produce a hard copy of your documents, this approach has considerable advantages over the use of a conventional word processor like OpenOffice or Abiword because the user creates plain text files which are portable across platforms. Moreover, although BSD-derived packages like **lpr** are more difficult to learn about than GUI based printer management tools, the Berkeley Unix (BSD) approach gives the user a fine-grained control over the printer.

10.8.2. Planning Ahead

Because of the complexity of the printer configuration process, we shall not discuss the topic in any detail. We would however, advise any user to do the following:

- Obtain your printer's technical specifications which will usually be available from the manufacturer's web site.

- Read through all relevant document - both HOWTOs and man pages - before doing anything.

- Check that your printer is working from within RISC OS before attempting to configure it for Linux.
- Do a test print using the lptest utility.
- Edit the `/etc/printcap` file where necessary.
- Ensure that Ghostscript works both with the X Windows System and through the shell.
- Use an application like magicfilter which helps you to configure your printer.

> **Tip:** If you have a Linux system up and running and you want to print man pages, an environmentally friendly method is to take advantage of the **psnup** command which forms part of the pstools package. This command allows you to decide how many man pages will be printed per page although it is dependent on the user being able to convert the requisite page to PostScript format.

10.8.3. A Simple Example

Assuming that you have successfully configured your printer, printing through the shell generally involves two stages:

1. the user creates a document with an editor like Vim or Emacs which may be interlaced with formatting instructions. Such formatting instructions are typically specific to either TeX or Groff.

   A number of filters may also be applied to the file if the user intends to do things like formatting line lengths. Other examples of typical filters include Enscript and a range of *magic* filters which convert to PostScript on the basis of a file's initial filetype.

2. The file is then sent to the printer by using **lpr**, Enscript or Ghostscript.

With regard to point two, **lpr** could be used to print a plain text file formatted according to personal preference. If a more complex document has been generated with either TeX or Groff which includes floating objects like tables and graphs, then one possible alternative would be to convert the file to PostScript as a prelude to

printing to a PostScript printer. Groff and TeX files can for example both be converted to PostScript using Dvips - a DVI to PostScript driver. Alternatively, users without a PostScript printer can also take advantage of Ghostscript- a program which can be used as a PostScript viewer, file converter and printer driver.

So in order to make the process more meaningful, let's use a fictitious document to illustrate the process in action. Suppose we have written a letter to a friend using LaTeX - the TeX macro set. For those who are unfamiliar with LaTeX, a sample file together with formatting instructions, is listed below:

```
\documentclass[a4 paper,12pt]{letter}
\name{Mr A. L. User}
\address{ARMLinux Users' Group\\
         Torvald's Road\\
         Linuxland}
\signature{ARMLinux User}
\date{June 12, 2000}
 \begin{document}
  \begin{letter}{Aleph One\\
                 The Old Courthouse\\
                 Bottisham\\
                 Cambridge}
\opening{Dear Aleph One}
I am writing to say how pleased I am with Aleph ARMLinux. Yes it
is very different from RISC OS but it's worth perservering with.
\closing{Yours sincerely}
  \end{letter}
 \end{document}
```

You would of course create this document with your favourite editor before saving it as `letter.tex`. Don't forget the suffix which indicates that the file is a LaTeX source file. `letter.tex` would then be processed by using the **latex** command to create a *Device-Independent* or DVI file - `letter.dvi` - with the *dvi* suffix.

latex letter

You are now ready to convert `letter.dvi` to PostScript by using Dvips. If you are successful, you will be left with `letter.ps`.

dvips -o letter.ps letter.dvi

And finally, print using either **lpr** Enscript or Ghostscript.

lpr letter

You can also print the raw LaTeX file with the **pr** command although this would not be particularly appropriate in a situation where one wished to process the formatting instructions as opposed to seeing them on the page.

pr letter

10.8.4. Resources

There is a wealth of information on printing and related utilities under Linux.

But your first port of call should be the:

Aleph ARMLinux CD

The printing HOWTOs are available on the CD in the `Docs.Html-Howto`. The relevant man pages are, as always, ever useful.

PostScript

General PostScript and Ghostscript resources are available from http://www.geocities.com/SiliconValley/5682/postscript.html/. PostScript-specific information can also be found on Adobe's web site at http://www.adobe.com/print/postscript/.

Ghostscript

Ghostscript's Home Page can be found at http://www.cs.wisc.edu/~ghost/ whilst the Ghostscript Manual is available from http://www.pdflib.com/gsmanual/index.html/.

TeX

Point your browser to http://www.tug.org/in order to learn about printing TeX-related files. Similarly, http://www.sunsite.ubc.ca/Resources/Graphics/dvips/ is the link to a comprehensive manual on the DVI to PostScript driver.

10.9. The X Windows System

10.9.1. The Client-Server Model

Although it is possible to work entirely through the shell, modern Linux distributions allow the user to work through a GUI and ARMLinux is no exception! Indeed, if your preferred method of working is to *point-and-click*, then you will feel very much at home with this distribution.

However, a brief explanation of the X Windows System is in order so that you can take advantage of the power which is at your fingertips.

The X Windows System or "X" as it is more commonly known is based on a client-server way of doing things in which the server program determines access to

your graphics hardware. The client, in contrast, is simply an applications program like xpdf or one of the many types of window manager. Although X clients usually run locally, remote connection to a client is possible and can be very useful in situations where you need to access an important file on another computer. RISC OS users will no doubt be interested to know that it is even possible to start up an X-Server under RISC OS itself if you take advantage of an application like !X (http://gnome.co.uk/) or !X-Server (http://kyllikki.fluff.org/software/x/).

10.9.2. Window Managers and X Applications

There are many window managers available for GNU/Linux so for reasons of brevity, we will confine ourselves to a description of Window Maker's basic features because this is the default desktop for Debian.

X Windows can be activated with the **startx** command. . To start Window Maker simply press the right mouse button and click on WindowManagers and then on the wmaker sub-menu.

When you fire up Window Maker, you will notice a number of distinctive features which include:

- The *Window Maker Workspace* from which you can launch applications by clicking with the right mouse button before choosing the application which you require.
- An *Applications Dock* on the right hand side of your screen which is used to activate applications which you use regularly.
- A series of *Miniwindows* at the bottom of your screen. The *Miniwindow* is made up of an icon and a title and appears when you *temporarily* close an application window.

To end your Window Maker session, click on the workspace with your right mouse button and select WindowManagers and Exit or Exit Session. If you choose Exit Session, all applications will be closed.

For further information, take a look at the official Window Maker web site (http://www.windowmaker.org/) which has links to mailing lists, user guides and sample configuration files where you wish to customise your setup.

Congratulations, you can now use all of the X applications which are on the CD but you will need to make a few changes to the default configuration before X Windows is fully usable.

- Add the line **`*input: true`** to the `.X` defaults file which will be in your home directory. This should solve the problem of being unable to type into dialogue

boxes under X.

- You will need to install pcnfsd if you wish to use the Network Filing System (NFS) for sharing files over a network to a RISC OS machine or a PC. It is not needed for Unix-Unix sharing. Supplementary information on this topic can be found on the CD in `Docs/html-howto`. The Networking HOWTO is also required reading.

So we've taken a look at Window Maker itself and in order to help you to use this window manager, what follows is a description of some of the core X applications which you may find useful. If you decide to install an application, follow the instructions in Section 10.4.4 which tell you how to install a package by using dpkg.

X Applications

xterm

A window containing the shell. xterm allows the user to cut-and-paste and to scroll. To close the xterm window, type **`exit`** at the prompt, press **Ctrl+D** or select Close with your mouse by clicking on the button on the extreme left of the fvwm titlebar.

xpdf

A viewer for Portable Document Format (PDF) files. xpdf has been ported to RISC OS by Leo Smiers.

xfig

A program which is ideal for the production of complex drawings which can be saved in a variety of formats including PostScript and LaTeX.

xv

Allows the user to view and convert images in a variety of formats including GIF, JPEG and TIFF. Facilities are also available for image cropping, rotation, gamma correction and for the capturing of screenshots.

games

You can even have some fun with Linux by playing games like xmahjonng, xgalaga or xgammon. If you want to have fun without using X, then why not check out bsd-games - a collection of classic non-X games including battleshar, gomoku and phantasia?

10.10. ARMLinux and the Internet

One of the attractions of ARMLinux is that there are so many tools which allow you to read and send news and email and to view web pages. This section will therefore show you how to establish a dial-up connection as well as introducing you to some useful net-related utilities.

10.10.1. Connecting to the Net with PPP

In order to get online you will need to set up your dial-up connection so we'll help you to do this by taking a look at the Point-to-Point Protocol (PPP).

This section has consequently been written for those who wish to gain a general sense of what is involved before consulting alternative resources. Newbies should therefore read this section before referring to the PPP-HOWTO which is on the CD in `Docs.Html-Howto`. Useful information can also be gained from the `README.linux` file in `/usr/doc/ppp-2.2.0f-2`, the pppd man pages and from the PPP FAQ (http://cs.uni-bonn.de/ppp/faq.html/). Users who wish to establish their own PPP server or to set up a Wide Area Network (WAN) will also find these resources helpful. Moreover, those who wish to use the Debian distribution of ARMLinux should refer to `install.en.txt` on the CD in `Debian.docs` for an explanation of pppconfig. Debian users will also find the wvdial dialler to be invaluable.

> **Note:** We have avoided any reference to SLIP connections or to DIP scripts because of the sheer complexity of the latter.PPP configuration will also become easier as ARMLinux is developed. The RHS Linux Network Manager is already available which allows users to configure interfaces, domains and name servers from the fvwm control panel and the eventual appearance of KDE will mean that PPP can be set up by using the kppp tool.

At present, you may set up your internet connection by:

a. Writing your own scripts.

b. Using the pppsetup program.

10.10.1.1. Writing Scripts

In order to configure PPP, you will need to find out the following information which is usually available from your Internet Service Provider (ISP):

1. Dialup number.
2. PABX number if you require an outside dialling tone.
3. Login details which will usually be a combination of letters and numbers.
4. Password-again a combination of letters and numbers.
5. Gateway-four numbers punctuated by dots.
6. Domain Name Server (DNS) Internet Protocol (IP) address - as with gateway.

> **Note:** It is advisable to ask for two DNS IP addresses in case one of the servers is temporarily disconnected from the net.

You will also need to ask your ISP about:

1. Its support for PAP or CHAP authentication.
2. The allocation of IP addresses. Nowadays, ISPs tend to *dynamically* assign IPs on a session-by-session basis. If this applies to you, this may have implications for how you offer server apps like ftpd on your machine.
3. The IP and *netmask* numbers that can be used in a situation where provision has been made for a subnet of IP numbers.

You will also have to think about configuring your modem which will involve:

1. Finding the name of the COM port.
2. Creating a link between the COM port name and `/dev/modem`.
3. Taking other factors into account like flow-control, line-termination and the connection speed that you will require.

> **Tip:** if you don't know the name of your COM port *pipe* the output of **dmesg** through **fgrep**:

```
dmesg | fgrep tty
```

If you require further information on configuring your modem, take a look at the Serial-HOWTO which is on the CD in `Docs.html-Howto`.

> **Note:** You may also need to recompile your kernel to enable PPP support.You can check that PPP support is enabled by availing of the **dmesg** command:

dmesg | fgrep PPP:

If you think for example of a situation where you want to establish a connection with Freeserve - a popular account with RISC OS users - PPP configuration can be achieved by taking the following steps:

- Edit the `/etc/ppp/options` file so that it contains information on the device that you will be using as well as the script - CHAP or PAP - that you will be using for dialling:

  connect ’/usr/sbin/chat -v -f /etc/ppp/chat-script’
  asyncmap 0
  crtscts
  defaultroute
  modem
  lock
  mtu 542
  mru 542
  115200
  /dev/modem

- Set up configuration and dialer files which may be found in `/usr/doc/ppp-2.2.0f` or in `/etc/ppp`. Assuming a chat script, enter the following in `/etc/ppp/chat-script`:

  ABORT ’NO CARRIER’
  ABORT BUSY
  ”ATF1
  OK ATD084550796699
  CONNECT”
  login:USER.freeserve.co.uk
  word:PASSWORD

- Create or adapt the *secrets file* if you are using PAP or CHAP authentication where both files - `/etc/ppp/chap-secrets` and `/etc/ppp/pap-secrets` - take the form of:

  USER.freeserve.co.uk*PASSWORD

- Run pppd as root.
- Configure `/etc/resolv.conf` so that domain names are associated with IP addresses:

  # /etc/resolv.conf

  search.

domain freeserve.co.uk
nameserver 194.152.64.35
nameserver 195.92.193.8

- Automate the connection process by copying the ppp-on and ppp-off scripts to `/usr/bin` as root. This will then allow you to toggle your connection *on* and *off* with the equivalent command.

10.10.1.1.1. Fine-Tuning

You should then

- Secure all passwords on your system by setting them to *660* with **chmod**.
- Log all PPP operations where you are having problems by entering the following command:

  tail -f /var/log/messages

- If you want to use PPP as a normal user, SUID pppd by entering this command at the prompt:

  chmod +s 'which pppd'

10.10.2. PPPsetup

The pppsetup program is an effective alternative for those who feel intimidated by the process of *hand-crafting* an internet connection. PPPsetup is available from http://members.nbci.com/chrissawer/ and was orginally derived from PPP version 2.3.5. Newbies should consequently note that the `chat` and `chat.8` files should be placed in `/usr/sbin` and `/usr/man/man8` respectively. The program also needs a current version of chat which can be downloaded from ftp://cs.anu.edu/pub/software/ppp/.

10.11. Shutting Down

Shutting down an ARMLinux machine is substantially different from what you are used to. Simply resetting the machine can for example lead to the corruption of data, or at least a very slow boot due to the disc-checking process. You can shut the

system down properly with the **halt** command although you will have to be logged in as root:

halt

If you want to reboot then you use **reboot** instead. Even easier is to use **Ctrl**-**Alt**-**Del** which also does 'reboot', even if you are not root. These are actually synonyms for various options to the **shutdown** command - i.e. **shutdown -h now** is the same as the **halt** command. There are of course many other options or *switches* that you can use in relation to this command, but you don't need to know those now. Use **man 8 shutdown** to read the details if you are interested.

10.12. A Word of Encouragement

This subsection has only scratched the surface of what can be accomplished with ARMLinux. We have however given you enough information to enable you to take those first tentative steps on your journey towards GNU/Linux enlightenment.

Good Luck and welcome to the open-source revolution!

Chapter 11. The ARM Structured Alignment FAQ

Q: What is structure alignment?

A: All modern CPUs expect that fundamental types like ints, longs and floats will be stored in memory at addresses that are multiples of their length.

CPUs are optimized for accessing memory aligned in this way.

Some CPUs:

- allow unaligned access but at a performance penalty;
- trap unaligned accesses to the operating system where they can either be ignored, simulated or reported as errors;
- use unaligned addresses as a means of doing special operations during the load or store.

When a C compiler processes a structure declaration, it can:

- add extra bytes between the fields to ensure that all fields requiring alignment are properly aligned;
- ensure that instances of the structure as a whole are properly aligned. Malloc always returns memory pointers that are aligned for the strictest, fundamental machine type.

The specifications for C/C++ state that the existence and nature of these padding bytes are *implementation defined*. This means that each CPU/OS/Compiler combination is free to use whatever alignment and padding rules are best for their purposes. Programmers however are not supposed to assume that specific padding and alignment rules will be followed. There are no controls defined within the language for indicating special handling of alignment and padding although many compilers like gcc have non-standard extensions to permit this.

Summary

Structure Alignment

> Structure alignment may be defined as the choice of rules which determine when and where padding is inserted together with the optimizations which the compiler is able to effect in generated code.

Q: Why is this an issue for ARM systems?

A:

- The early ARM processors had limited abilities to access memory that was not aligned on a word (four byte) boundary.
- Current ARM processors (as opposed to the StrongARM) have less support for accessing halfword (short int) values.
- The first compilers were designed for embedded system applications.
- The compiler writers chose to allow the declaration of types shorter than a word (chars and shorts) but aligned all structures to a word boundary to increase performance when accessing these items.

These rules are acceptable to the C/C++ language specifications but they are different from the rules that are used by virtually all 32 and 64 bit microprocessors. Linux and its applications have never been ported to a platform with these alignment rules before so there are latent defects in the code where programmers have incorrectly assumed certain alignment rules. Moreover, these defects appear when applications are ported to the ARM platform.

The Linux kernel itself contains these types of assumptions.

These latent defects can consequently lead to:

- decreased performance;
- corrupted data;
- program crashes.

The exact effect depends on how the compiler and OS are configured as well as the nature of the defective code.

These defects may be fixed by:

1. changing the compiler’s alignment rules to match those of other Linux platforms;
2. using an alignment trap to fix incorrectly aligned memory references;
3. finding and fixing all latent defects on a case-by-case basis.

The three alternatives are, to some extent, mutually exclusive. All of them have advantages and disadvantages and have been applied in the past so there is some experience with each although the correct solution depends on your goals (see below).

Q: How is this related to the alignment trap?

A: On the StrongARM processor, the OS can establish a trap to handle unaligned memory references. This is important because unaligned memory references are a frequent consequence of alignment traps although they are not the only consequence.

Thus, some, but not all, alignment defects can be fixed within an alignment trap.

Furthermore, not every unaligned access indicates a defect. In particular, compilers for processors without halfword access will use unaligned accesses to efficiently load and store these values. If the alignment trap *fixes* these memory references, the program will produce incorrect results.

On the ARM and StrongARM, if you ask for a non-aligned word and you don't take the alignment trap, then you get the aligned word rotated such that the byte align you asked for is in the LSB.

Consider:

```
Address: 0  1  2  3  4  5  6  7
Value  : 10 21 66 23 ab 5e 9c 1d
```

Using *(unsigned long*)2 would give:

```
on x86: 0x5eab2366
on ARM: 0x21102366
```

An alignment trap can distinguish between kernel code and application code and do different things for each.

The basic choices for the alignment trap are:

1. It can be turned off. The unaligned access will then behave like unaligned accesses on other members of the ARM family without performance penalty.
2. It can "fixup" the access to simulate a processor that allows unaligned access.
3. It can fixup the access and generate a kernel message.
4. It can terminate the application or declare a kernel panic.

There is a significant performance penalty for fixing up unaligned memory references.

Q: Which compilers are affected?

A: The ARM port of GCC can be configured to either:

1. align all structures to a word boundary -- even those containing just chars and shorts (this is the way ARMLinux is distributed);
2. align structures based on the alignment constraints of the most strict structure member (this is the same alignment as is used on the x86);
3. follow other rules.

Changing between *1* and *2* is a one line change in the gcc or egcs source. With additional effort, these could be modified with an additional compile time parameter selecting the alignment rules to be used. Some other architectures already have such a flag, so these could be used as a model.

The compiler supplied with the ARM SDT defaults to align all structures on a word boundary. It has a "packed structure option" (`-zas1`) that changes alignment to match the x86 rules. In future, this option will be the default since:

> word-alignment causes too much user trouble, and the performance/codesize improvement has never been proven (typically the affected structures are small, and generally not copied around a lot).

Q: What are the advantages of word structure alignment?

A: The advantages are as follows:

- structure copies for structures containing only chars and shorts are much faster;
- faster code can be generated for halfword access on pre ARM.v4 processors;
- binary compatibility across all ARM processors;
- binary compatibility with the original compilers;
- compatibility with ARM Directives.

The overall performance impact on StrongARM processors is hotly debated.

Here are some typical responses:

> unattributed
>
> most of the system would run faster

> unattributed
>
> pretty much ANYTHING that you care about memory bandwidth and performance issues on will or could seriously be impacted by this.

> unattributed
>
> Although in theory it produces faster code, in practice most code and thus the system will run a lot slower.

The performance impact on other processors is less debated, but there is not complete consensus there either.

The only way to resolve this debate is to measure the relative performance.

Q: What are the disadvantages of word alignment?

A: The disadvantages of word alignment are that:

- it exposes latent defects. If the compiler aligned consistently with other Linux platforms, these defects would remain latent and the effort involved in porting Linux to the StrongARM would be reduced;
- uncorrected defects can silently corrupt data;
- uncorrected defects cause unreliability. Alignment defects that only show up under heavy load, under certain patterns of use, at particular optimization levels, or in certain configurations are particularly difficult to find and fix;
- the corrected programs are "fragile". Subsequent code changes can create or expose new alignment defects;
- it effectively decreases performance.

There is hot debate on both the number of Linux packages that have latent alignment defects and how difficult these defects will be to find and fix. Estimates of the magnitude of the problem include:

> unattributed
>
> The only programs that I found that were violating this when I did the original port were very few and far between. I think it was in the order of 1 in 200. However, as of lately, maybe because of the commercialisation of the Internet, this figure appears to be increasing.

> unattributed
>
> Generally, the defects I’ve found stick out like a sore thumb.

> unattributed
>
> These problems are so severe that I’d be very surprised if any major Linux application runs reliably or can be made to run reliably without superhuman effort.

Unless other measures are taken, this debate will not be resolved until ARM distributions that align all structures are complete and widely deployed or the attempt is abandoned. Distributions that elect to not align all structures avoid the problem and thus never find out its magnitude in detail.

The alignment trap for application code can be used to produce an estimate of the problem magnitude earlier than this. Application code will execute unaligned memory references in the following circumstances:

- it was compiled for an ARM processor and is using "legal unaligned load word instructions" to reference halfword and/or byte data;
- the application has code that deliberately does unaligned memory references. This indicates that the application is not portable to a variety of platforms;
- the application has latent alignment defects exposed by alignment rule differences.

When the alignment trap is set to generate a count of traps from application code and code compiled for the StrongARM is run, then every trap signals the existence of a defect that needs to be fixed. If the problem magnitude is large, many messages/counts will be recorded. If the problems are rare or have already been fixed, the trap will be silent.

The early results of this testing on the NetWinder have been:

- X generates literally tens of millions of alignment faults. Line drawing is particularly nasty;
- without X, after a reset and login there are 21256 userland faults;
- GCC generates unaligned load/store multiple instructions now and then too.

This picture has changed as more and more packages are updated to newer versions and compiled with newer compiler versions to the point that the number of traps has declined to about 1,000 per CPU minute even with X windows use.

Setting the alignment trap to produce messages or counts is obviously useful for debugging as well. However, it produces only an estimate of the magnitude because there are potential latent defects that will cause applications to fail without ever doing an unaligned memory reference.

The argument that aligned structures are effectively slower is based on three positions:

1. the fixes to alignment defects often result in slower code;

2. the alignment trap would be called less frequently if the compiler didn’t align all structures;
3. code compiled for ARM processors will execute slower than code compiled specifically for the StrongARM.

Q: What is the magnitude of the porting problem?

A: At this point, several years of fixing alignment defects in Linux packages have reduced the problems in the most common packages.

Packages known to have had alignment defects are:

- Linux kernel;
- binutils;
- cpio;
- RPM;
- Orbit (part of Gnome);
- X Windows.

This list is *very* incomplete.

Q: Why can’t we just change the compiler?

A: The problem with changing the compiler is one of compatibility and transition. A completely new distribution for the ARM or StrongARM could use whatever alignment rules meet its goals. However, there would be problems running binaries from other distributions. For commercial applications, this would split the ARM market in two and they would need to decide which distribution(s) to support.

Those familiar with Unix history know the potential costs of these splits.

Since StrongARM binaries cannot be run on the ARM processors, this is the natural dividing point for this split. To some extent, this split has already occurred since many packages are being ported specifically for the StrongARM.

Changing alignment in a StrongARM distribution will therefore affect its ability to run ARM binaries.

From this perspective, the worst case is having two binary standards on the StrongARM processor for the same OS.

The upgrade from aligned to unaligned or vice versa is particularly tricky because of interdependencies between programs and shared libraries. When the upgrade is in progress, the system is really some kind of "mixed distribution". Also, local

programs compiled before the upgrade need to be recompiled to ensure compatibility.

Q: What about mixed distributions?

A: It is possible to create header files for libraries and system calls that are independent of which alignment rules are used by the compiler and thus ensure binary compatibility between distributions even if different compilers are used. All of the distributions would need to standardize on these modified headers for this to work.

If these changes were in place, different applications could be compiled with different rules within the same distribution as the needs of the application itself dictate. Some people are going to be experimenting with alternatively configured compilers and will need to make at least a start on these changes in order to do this experimentation. Later in this FAQ is a list of the system header files that would be affected.

Q: Some examples of code with problems?

A: All of the following examples are defective in a way that works for most Linux platforms and fails under the ARMLinux distribution. The behaviour of the ARMLinux distribution is described.

Example A

Suppose, I'm doing something to a truecolour image in C++ (brightening it for instance) and I have a pointer to the image in memory.

```
struct Pixel
{
        unsigned char red;
        unsigned char green;
        unsigned char blue;
};

unsigned char* image;

Pixel*  ptr = (Pixel*)image;

inline brighten(Pixel* pix)
{
        //...a bunch of code that references *pix
}

for (int x=0; x>1024; x++)
```

```
{
        brighten(ptr++);
}
```

The Pixel structure will be padded with an extra byte at the end and will be aligned to a word boundary. Each ptr++ will step the pointer by four bytes instead of three as intended and thus the image will be corrupted. If image is aligned on a word boundary (this is random chance), no unaligned memory references will be made.

If the loop is alterred so that ptr is incremented by three bytes instead of four, then the image may be corrupted depending on what brighten does and the optimization level.

Example B

Suppose now, I have an alpha field:

```
struct RGBAPixel
{
        unsigned char   alpha;
        Pixel           pxl;
};

This is an 8 byte structure with a layout totally different from

struct RGBAPixel
{
        unsigned char   alpha;
        unsigned char   red;
        unsigned char   green;
        unsigned char   blue;
};
```

which is the layout on most other Linux platforms.

Example C

```
struct Date
{
        char    hasHappened;
        char    year[4];
        char    month[2];
        char    day[2];
};

struct Record
{
        char    name[20];
        Date    birthday;
```

```
        Date    marriage;
        Date    death;
        Date    last_taxes_paid;
} inbuf;

#define RECORD_LENGTH (20+4*9)

read(fd, &inbuf, RECORD_LENGTH);
```

All of the date fields will be corrupt after the read.

Example D

This example is from the Kernel source.

```
struct nls_unicode {
        unsigned char uni1;
        unsigned char uni2;
};

static struct nls_unicode charset2uni[256] = {...};
```

Each unicode character consumes four bytes instead of two as on other platforms. Although in this case, the only impact is benign (extra memory consumption).

Attempting to read, write, or copy unicode strings based on this definition would lead to problems.

Q: How do I find alignment problems in code from other platforms?

A: This section is fairly specific to ARMLinux application porting. Fixing all alignment problems, including those that may cause problems in future or on other platforms, is beyond the scope of this FAQ.

The gcc compiler for the ARMLinux distribution aligns all structures containing ints, longs, floats and pointers in the same way as gcc on x86 and other 32 bit platforms. The differences that may result in exposing latent alignment defects are all related to structures consisting entirely of chars and shorts either signed or unsigned. On ARMLinux, these are aligned to a word (4 byte) boundary. On other platforms these are aligned to a character boundary (i.e.: unaligned) for structures containing only chars and a halfword boundary for structures containing shorts or shorts and chars.

In practice, structures of this nature are relatively rare, so this is a good place to start looking.

The uses of these structures that may cause problems are:

- one of these structures is contained within another structure. This will generate additional interior padding in the containing structure unless the contained structure just happens to be already aligned. This interior padding will cause problems if the structure is read or written as whole or aliased to another data structure;
- an array of one of these structures or a containing structure is defined. These will be larger on ARMLinux unless the structure just happens to be a multiple of 4 bytes long. This is really just another type of internal padding and the same caveats apply;
- a pointer to something else (usually char* or void*) is cast to a pointer to one of these structures, a containing structure or an array of one of these. The compiler expects that all pointers to one of these structures are aligned on a word boundary. If not, the generated code can incorrectly load field values and silently corrupt memory contents on stores. The exact behaviour depends on how the fields are accessed and optimizations made by the compiler;
- sizeof() is taken on the structure, an instance of the structure or a containing structure or an array of one of these. The length returned will be different on ARMLinux unless the structure just happens to be an even multiple of 4 bytes long. This isn't a problem in itself, but the program may use this value inappropriately;
- sizeof() is assumed. Look for #defines and comments that mention the size of the structure, a containing structure or the array as these indicate potential problems;
- in a mixed environment (using different compilers or a compiler switch), problems occur if a structure or pointer to a structure is passed between a caller and a called routine that have different alignment rules or different interior padding. In this case, the mere existence of structures that would be differently aligned or padded in a public header file is a defect.

Q: How do I fix alignment problems?

A: This really depends on your goals.

If you are concerned with the long term portability of the code, you will find and remove all expectations about padding and alignment from it. How to do this is beyond the scope of this FAQ.

If you want to port a package to ARMLinux with minimal code changes or you suspect alignment problems and want a quick test, using the gcc extension __attribute__((packed)) will help in many cases.

If you want to arrange the header files of a library for binary compatibility between different alignment settings on StrongARM compilers, use __attribute__((packed)),

explicitly insert padding bytes, and/or force alignment with unions or zero length arrays.

Q: What about C++?

A: A C++ class is an extension of a struct and many of the same comments apply. In addition, inheritance and template classes introduce new ways of combining structures that can cause interior padding that is different between on ARMLinux and x86 systems. Name mangling may or may not be affected. This makes the problems more difficult to identify from the source code. C++ programs in particular need to be devoid of all expectations of interior padding and alignment.

Chapter 12. ARM devices and projects

This chapter contains information on the various ARM machines available around the world which are using ARMLinux.

12.1. ARM Devices

12.1.1. Aleph One Limited - The ARMLinux Specialists

Note: Aleph One currently supply ARMLinux distributions and documentation covering the Risc PC, LART, Assabet/Neponset and Psion 5. The Documentation also covers the generic principles for many ARM devices. Where a device is not listed, Aleph One are able to get ARMLinux up and running on your hardware on the understanding that you supply the necessary kit and hardware documentation. We will also take care of passing back the necessary changes to the main kernel tree.

12.1.2. Acorn and RISC OS Machines

Aleph ARMLinux is currently available for Risc PC users although there are alternative RISC OS machines for which support exists in the Linux kernel. These include:

- The RiscStation - this support is now functional but not yet quite ready for release. Watch the web sites for updates.
- The Acorn Archimedes range of computers which include machines like the A3000 and
- The A5000 series.

With regard to Linux installation on the Archimedes range, detailed information can be found on the ARMLinux web site. Alternatively, Chris Pringle’s manual - available from http://latrigg.demon.co.uk/chris/linux.html/ - provides a useful resource for those who wish to install ARMLinux on an 8MB A5000.

12.1.3. Psion

ARMLinux is also available in proof-of-concept form for the Psion Series 5 and for the related Geofox PDA because of the efforts of the Linux 7k project. ARMLinux for Psion uses the standard 2.2.1 Linux kernel with the addition of a specific psion-arm patch. This port offers support for established Psion and Geofox-related drivers and is able to take advantage of a range of applications. Only very limited use of Linux can be made without the addition of a compact flash card to extend the available storage.

Work is also in progress on a suitable Graphical User Interface of which an interesting example is the API developed by the MicroWindows Project. MicroWindow's API is known as Nano X and is much like xlib. At the time of writing (November 2000), this is just becoming functional on the Psion.

> **Note:** Aleph One Limited hope to have an ARMLinux distribution available for the Psion Series 5 by early 2001.

Distinctive features of ARMLinux for Psion include the

- Development of ArLo or the ARMLoader which loads and executes Linux whilst Psion's EPOC OS is running.
- Use of the Initrd virtual filing system image which contains a miniature version of Linux.

Additional information about the Linux 7k project is available from http://linux-7110.sourceforge.net/. Current information on how the project is progressing can similarly be obtained by joining the mailing list <`linux-7110-request@sourceforge.net`> which can be accomplished by writing "subscribe" in the subject area of your email. Linux 7k also maintain an ftp site at ftp://ftp.sourceforge.net/linux-7110 although the mirror site at Imperial College London - ftp://ftp.sunsite.org.uk/Mirrors/sourceforge.net/linux-7110 - is more convenient for European users. A comprehensive Linux 7k mini-HOWTO by Stephen Harris is similarly available from the project's web site.

Insofar as supplementary information about Geofox is concerned, the company web site is no longer available although hardware information is obtainable from ftp://ftp.linuxhacker.org/pub/geofox.

In a similar vein, up to date information on the Nano X project can be had by subscribing to the Nano X mailing list on

<nanogui-subscribe@linuxhacker.org> or by pointing your browser to http://microwindows.censoft.com/.

12.1.4. Footbridge Machines

Footbridge is a term which applies to DC21285- the PCI Core Logic device for the StrongARM processor. DC21285 appears in the NetWinder, in EBSA285 and in the CATS board.

12.1.4.1. NetWinder

Rebel.com's NetWinder range of computers include machines like the Office Server. This machine has been designed with the needs of home and small business users in mind who have need of a firewall/web/mail server which runs on Linux.

Distinctive features include support for:

- Secure connection outside of a LAN via Progressive Systems's Smartgate VPN server.
- Customisation of the user environment using specifications which are in the public domain.
- Easy access to programs from http://www.netwinder.org/.
- A default set of firewall rules which are useful in the prevention of IP spoof attacks.

12.1.4.2. EBSA285

EBSA285 or the Intel StrongARM SA-110/21285 Evaluation Board is a development environment for applications which will be based on the SA110 processor and 21285 Core Logic Chip.

EBSA285 can run Linux in addition to other OSs and is able to take advantage of a range of software from ARM and Cygnus.

Supplementary information on EBSA285 in the form of a product brief is available as a PDF file from the Intel Developers site at http://developer.intel.com/design/strong/quicklist/eval-plat/sa-110.htm/.

12.1.5. SA-1100

12.1.5.1. LART

The Linux Advanced Radio Terminal (LART) is a compact energy efficient computer which was developed in order to conduct experiments in wireless multimedia. LART's standard configuration includes 32MB DRAM in addition to 4MB Flash ROM which is sufficient for the Linux kernel and a RAMdisk image. Setting up Linux on the LART is relatively straightforward because of the existence of the necessary patches in the main SA-1100 Linux patch. However, users will also need to acquire some LART-specific patches which are available as tarballs from http://www.lart.tudelft.nl/download.php3/. One such patch is the clock scaling patch which allows the user to customise the clock speed of the SA-1100 CPU.

Other distinctive features include:

- Small size - 100mm x 60mm.
- A modular design which is easily updatable.
- Blob - LART's boot loader.
- The availability of a C/C++ cross-compiler.

The original developers at the University of Delft (TU) don't currently produce LARTs other than as university research tools so interested readers should consider joining the LART mailing list at <`lart@lart.tudelft.nl`> for news of developments. Conversely, CAD files are available for use as is the LART software and LART-specific patches. To subscribe to the LART mailing list, send a message to <`major-domo@lat.tudelft.nl`> with the phrase "subscribe lart" in the message body. Alternatively, archives of the mailing list are available from http://www.lart.tudelft.nl/list/list.php3?arc=lart/. For general information on the project go to LART's web site at http://www.lart.tudelft.nl/.

12.1.5.2. PLEB

This project - originally inspired by the Turbo Tortoise - encompasses a number of hardware and software initiatives together with associated tools which will be designed specifically for the aforementioned projects. PLEB was the brain-child of designers at the University of New South Wales http://cse.unsw.edu.au/~pleb/ who wanted to make a pocket computer which was able to run the Linux kernel.

Distinctive components of the PLEB project include the:

- Development of the Photon - a credit-card sized StrongARM prototype for embedded and/or portable applications.

- Atmel AVR board known as the Nova which has been used for small projects like the JTAG programmer.
- Catapult - a firmware boot loader which will replace Blob - the boot loader of the LART project.
- Gauntlet - an L4 micro-kernel for ARM.
- Anvil - JTAG software for hardware programming and diagnostics.

Supplementary information on the PLEB project can be obtained by subscribing online to the PLEB mailing lists which include the general list <pleb@cse.unsw.edu.au> or the hardware-specific list <pleb.hardware@cse.unsw.edu.au>.

12.1.5.3. Assabet

An SA-1110 evaluation board which may be repackaged as a PDA. Notable features include:

- The availability of Win32 programming utilities like the SA-1110 Development Board, JTAG Programming Software V0.3 and Jflash-Linux which have been ported to Linux by Nicholas Pitre.
- Neponset - a SA-1111 Development module which can deal with system lock ups which may in turn be caused by interrupt handling difficulties.
- Use of the 2.4 kernel series with support for the Assabet and Neponset via a patch chain.
- Alternative boot methods - the Angelboot utility is one example.
- The capacity to run MicroWindows although a backlight power source needs to be added.

Work is also underway on support for Compact Flash on the Assabet and on PCMCIA support for the Neponset (http://www.compactflash.org/). In addition, the Socket LP-E CF + Card is now operative (http://socketcom.com/lpecfmp.htm/) as is the IBM Microdrive (http://www.storage.ibm.com/hardsoft/diskdrdl/micro/).

12.1.5.4. Simputer

A project which is known as the Simple Inexpensive Multilingual Computer (Simputer http://www.simputer.org/). The Simputer exists to produce a low-cost Assabet-like PDA.

Distinctive features include:

- A smart card reader.
- Use of text-to-speech audio facilities.
- Use of a browser which renders Information Markup Language (IML) which is smart card aware.
- Touch screen and a local-language software interface.

More information is available from the Simputer mailing list which you can either join online (http://www.simputer.org/ or by writing “subscribe” in the body of an email which should be addressed to <simputer@egroups.com>.

12.1.5.5. Itsy

A pocket computing initiative which uses Linux and the standard GNU tools to offer the following distinctive features.

- Flexible interfaces for adding custom-made daughter cards.
- Support for speech recognition.

Additional technical information on the Itsy Pocket Computer and on the Memory Daughter Card - both for Version 1.5 - can be downloaded from the Itsy web site at http://research.compaq.com/wrl/projects/itsy/index.html/. A number of slide shows are also available from the same web site which showcase the project’s accomplishments to date.

12.1.5.6. The SmartBadge Project

The basic project - initiated by the Swedish Royal Institute of Technology http://www.it.kth.se/edu/gru/Fingerinfo/telesys.finger/Mobile.VT98/badge.html/ - uses an Intel StrongARM processor, sensors, built-in communication links and a PCMCIA interface. The complete schematic for this device is available from the SmartBadge web site as are a series of PDF files.

Particular features of the SmartBadge III Project include:

- Speech synthesization.
- giveio.sys - device driver for NT 4.0.
- instdrv.zip - a transient device driver which allows the user to load and start a device driver without necessarily having to permanently install it.
- tar.zip - a miscellaneous collection of TAP master code.
- TAP master documentation which is available from the SmartBadge web site

12.1.5.7. The Compaq Cambridge Research Lab Project

A StrongARM-based embedded computing research prototype which is intended for use with a variety of applications http://crl.research.compaq.com/projects/personalserver/default.htm/. One recent use of the Personal Server was in the form of a Compaq Robot Controller at MIT's Autonomous Robot Design Competition (http://mit.edu/6.270/). The Compaq Project is also noteworthy because of its use of a Debian based Linux distro - kernel 2.2.9 - from Chalice Technology (http://www.chaltech.com/. The distro currently fills an 8MB filesystem and therefore needs a remote connection to a filesystem in order to take advantage of additional applications.

A range of documents are available which are pertinent to this project including:

- *The Personal Server Users' Manual* which examines bootldr, Linux and NetBSD.
- Documentation for the Java package for the Personal Server with RCC.

There are also Compaq Server mailing lists at <robot-controller@crl.dec.com> and <6.270-organizers@mit.edu>. The latter list focuses on the MIT 6.270 contest.

Developers may also be interested in Cameron Moreland's HOWTO which is specifically concerned with running Linux on an Intel Brutus Board although it also contains much of relevance to those who intend to install Linux on a range of ARM machines. This document is available from the Aleph One web site http://www.aleph1.co.uk/armlinux/resources.html/.

Chapter 13. Sources of Help and Information

There are various places you can get help which include:

13.1. This Guide

This includes specific information relating to a selection of ARMLinux devices and generic info on tools and techniques.

Our web site will also grow to contain much useful ARMLinux information.

13.2. The ARMLinux Web sites

http://www.arm.linux.org.uk/ will take you to Russell King's main ARMLinux site which is the definitive source for current ARMLinux kernel updates, ARM architecture information and general info.

Aleph ARMLinux updates including new packages not yet available from Debian's standard distribution are available from ftp://www.armlinux.org/debian/. You should add this to your `/etc/apt/sources.list` file so that the latest updates will automatically be obtained when you do **apt-get upgrade**.

13.3. Debian Resources

The Debian Project (http://www.debian.org/) provides a number of guides and mailing lists which cater for different audiences. Reference has however only been made to documentation which is complete and up to date. Interested readers should consequently refer to The Debian Documentation Project's web page (http://www.debian.org/doc/ddp/) for details of work in progress.

13.3.1. Debian Guides

Debian Developers' Reference

http://www.debian.org/doc/packaging-manuals/developers-reference/

Debian FAQ-O-Matic

http://www.debian.org/cgi-bin/fom/

Debian GNU/Linux: Guide to Installation and Usage

http://www.newriders.com/debian/html/noframes/

Debian New Maintainers' Guide

http://www.debian.org/doc/maint-guide/

Debian Packaging Manual

http://www.debian.org/doc/packaging-manuals/packaging.html/

Debian Policy Manual

http://www.debian.org/doc/debian-policy/

dselect Documentation for Beginners

ftp://ftp.debian.org/dists/stable/main/disks-*/current/doc/dselect-beginner.*/

13.3.2. Debian Mailing Lists

There are a number of mailing lists for this Linux distribution but <debian-user@lists.debian.org> is a good place to direct your questions. To subscribe, type "subscribe" in the subject line and mail to <debian-user-REQUEST@lists.debian.org>

In order to find out about other mailing lists, take a look at `mailing-lists/txt` which is on the Aleph One CD in the `Doc` directory.

You can also refer to Section 13.7 for details of ARM-specific lists.

13.3.3. Debian Help

You can similarly learn how to solve Debian-specific problems by pointing your browser to http://www.debianhelp.org/.

13.4. News Groups

There are many news groups which provide useful information which is not strictly speaking ARMLinux specific but which is none the worse for this including:

- comp.os.linux.advocacy which will keep you in touch with current software releases.
- comp.os.linux.answers which provides the definitive source of information for current HOWTOs and FAQs.
- uk.comp.os.linux which is a useful source of help.

13.5. The `Doc` folder on this CD

This directory contains information which will be of interest to users and developers including:

`FAQ/`

The Debian/GNU Linux FAQ in HTML, PostScript and Plain Text formats.

`constitution.txt`

The Debian Constitution.

`debian-manifesto`

How it all started...

`mailing-lists.txt`

How to subscribe to the Debian mailing lists.

`social-contract.txt`

Describes Debian’s commitment to free software.

`source-unpack.txt`

How to unpack a Debian source package.

`debian-keyring.tar.gz`

Developers’ Pretty Good Privacy (PGP) keys.

`bug-reporting.txt`

How to report a bug in Debian.

`package-developer/`

Documentation for developers.

13.6. Aleph One Technical Support

If you bought this book with a CD direct from Aleph One then it comes with technical support. Please email first, but if things get complicated than a phone call may be useful, quoting your registration code, and we will do our best to help. This service is currently focused on helping you to get ARMLinux installed and set up; queries about other aspects of Linux are probably best directed elsewhere.

The support email address is <armlinux-support@aleph1.co.uk>. The support phone number is 01223 811679.

13.7. Mailing Lists

There are now quite a few, and which one(s) are most appropriate, depends on your level of experience and area of interest.

13.7.1. linux-arm

<linux-arm@lists.arm.linux.org.uk>. This is the main ARMLinux list which covers ARMLinux topics not specific to and of the other mailing lists. This covers all architectures that ARMLinux is used on (Desktops, PDAs, development boards etc), so much of the discussion is not specifically related to your machine. Users who have ceased to be *newbies* will find this of interest. To subscribe send "subscribe" (in the body of the message) to <linux-arm-request@lists.arm.linux.org.uk>, or use the web interface at:http://www.arm.linux.org.uk/armlinux/mailinglists.shtml.

13.7.2. linux-arm-kernel

<linux-arm-kernel@lists.arm.linux.org.uk> is for kernel-related discussions, generally of a technical nature. This normally covers the kernel, related tools and new hardware. If you are a Linux beginner you won't understand a word of what goes on here :-) This is the place for those interested in the grubby stuff. To subscribe send "subscribe" (in the body of the message) to <linux-arm-kernel-request@lists.arm.linux.org.uk>.

13.7.3. linux-arm-announce

<linux-arm-announce@lists.arm.linux.org.uk> is a moderated low-traffic list for announcements relevant to ARMLinux. To subscribe send

"subscribe" (in the body of the message) to <linux-arm-announce-request@lists.arm.linux.org.uk>.

13.7.4. armlinux-toolchain

<ARMLinux-toolchain@lists.armlinux.org>This list serves mainly as a place for developers to discuss any aspect of the GNU development toolchain on the ARM architecture (including other operating systems). It's also a diagnostic forum to an extent, if you're having problems. http://www.armlinux.org/cgi-bin/mailman/listinfo/armlinux-toolchain and fill in the form. An archive of submissions to this group is also available from http://www.armlinux.org/pipermail/armlinux-toolchain/

13.7.5. ARMLinux-newbie

There is a mailing list specifically for people new to ARMLinux, where you are positively encouraged to ask *stupid questions*. This is aimed at users as opposed to developers, particularly RISC OS machine users and PDA users , and is at <ARMLinux-newbie@lists.armlinux.org>. This is not the place to ask about toolchains, the kernel, or any development-type questions. This is the place for installation problems, and questions about software that just crashes or doesn't seem to work right. We recommend that all users join this list so that the community can help each other out and compare notes. An archive of submissions to this group is also available from http://www.armlinux.org/pipermail/armlinux-newbie/.

13.7.6. Debian ARM

If you are using, or working on, the Debian ARM distribution then <debian-arm@lists.debian.org> is the place to talk to other users and developers about any ARM-specific issues. Debian has loads of maiing lists for other aspects of the distribution.

13.7.7. Emdebian

<emdebian-discuss@lists.sourceforge.net> is a project to make Debian work on small embedded systems and PDAs. It supplies a very good Toolchain and experimental software (CML2+OS) to generate small installations and kernels from the huge Debian archive. It needs some more active developers - join this list if you think you can help or are interested in the design process.

13.7.8. LART

<lart@lart.tudelft.nl> is the place for all tings LARTy. All LART users should be subscribed here. Its the place to get hardware or software help and discuss the possibilities of your LART.

13.7.9. Psion PDAs

<linux-7110@sourceforge.net> is concerned with the Psion 5/5mx/7 port of ARMLinux and is the primary channel for information. This is a list both for developers and users. To subscribe, go to lin7kmailURL;.

13.7.10. StrongARM issues

<sa1100-linux@pa.dec.com> is for those who are interested in StrongARM-specific issues - primarily kernel development.

13.7.11. iPAQ PDAs

The iPAQ has proved a very popular ARM/PDA development platform due to Compaq's support. There is an active community working on distributions and the hardware on this list: <ipaq@handhelds.org>. Handhelds.org also has other lists, not necessarily specific to ARM.

13.8. Reading List

There are many good books about Unix and Linux. None of them are aimed specifically at ARMLinux but most will nevertheless be helpful. A list of books and ISBNs which we have found helpful is provided in the bibliography.

13.8.1. Books Online

A number of commercially available books and sample chapters are also available online including:

Advanced Linux Sound Architecture

http://sourceforge.net/projects/alsa/

ARM Linux Project

http://www.arm.linux.org.uk/

Learning Debian GNU/Linux

http://www.oreilly.com/catalog/debian/chapter/index.html/

Learning the Vi Editor

http://www.oreilly.com/catalog/vi6/chapter/ch08.html/

Linux Console Project

http://sourceforge.net/projects/linuxconsole/

Linux Framebuffer Driver Writing HOWTO

http://www.linux-fbdev.org/HOWTO/

Open Sound System

http://www.opensound.com/

The Unix CD Bookshelf

http://www.oreilly.com/catalog/unixcd/chapter/index.html/

Bibliography

How to Use Linux: Visually in Full-Color, Bill Ball, 0672315459, Sams Publishing, 1999.

Debian GNU/Linux for Dummies, Michael Bellomo, 0764507133, IDG Books Worldwide Inc, 2000.

Learning GNU Emacs, Debra Cameron, Bill Rosenblatt, and Eric Raymond, 1565921526, O'Reilly & Associates, 1996.

Debian GNU/Linux 2.1 Unleashed, Aaron van Couwenberghe, 0672317001, Sams Publishing, 1999.

Installing Debian GNU/Linux, Thomas Downs, 0672317451, Sams Publishing, 1999.

The Unix CD Bookshelf, Daniel Gilly, 1565924061, O'Reilly & Associates, 1998.

Debian GNU/Linux, John Goerzen and Ossama Othman, 0735709149, New Riders, 1999.

Debian GNU/Linux Bible, Steve Hunger, Randolph Chung, and Steve Chung, 0764547100, IDG Books Worldwide, 2000.

Learning Debian Linux, Bill McCarty, 1565927052, O'Reilly & Associates, 1999.

Linux Device Drivers, Alessandro Rubini and Jonathan Corbet, 2nd Edition, 0596000081, O'Reilly & Associates, 2001.

The Debian Linux Users' Guide, Dale Scheetz, 0965957519, Linux Press, 1998.

Beginning Linux Programming, Richard Stones and Neil Matthew, 2nd Edition, 1861002971, Wrox Press, 1999.

Running Linux, Matt Welsh, Matthias Kalle Dalheimer, and Lar Kaufman, 3rd Edition, 156592469X, O'Reilly & Associates, 1999.

Chapter 14. Feedback

Aleph One will be monitoring the various ARMLinux mailing lists, and will improve this distribution according to the sorts of requests and problems we see.

We are also keen to receive email on <devel-armlinux-feedback@aleph1.co.uk> with suggestions for improvements.

And you can write to us at Aleph One Ltd, The Old Courthouse, Bottisham, CAMBRIDGE, CB5 9BA, UK

Appendix A. Jed: Simple Commands

Key Combination	Function
C-a	move cursor to beginning of line
C-b	move back one character
C-d	delete a character
C-e	move cursor to end of line
C-f	move forward one character
C-k	delete the line
C-n	move cursor to next line
C-p	move cursor to previous line
C-v	page down
C-x u	undelete
C-x C-c	exit
C-x C-f	find a file
C-x C-s	save file
C-x C-v	find alternative file
C-x C-w	write to a file with new filename
ESC b	back one word
ESC d	delete word
ESC f	move cursor by one word
ESC <	move to beginning of file
ESC >	move to end of file
ESC v	page up

Appendix B. Vim: Simple Commands

Key	Function
b, B	back one word
cc	change current line
CTRL-B	scroll one screen back
CTRL-F	scroll one screen forward
CTRL-G	show current line
cw	change word
:d	delete a character
:dd	delete current line
:dw	delete a word
:/expression	forward search for expression
:?expression	backward search for expression
:e!	wipe session edits
h, j, k, l	left, down up or right
i, a	insert text before/after cursor
p, P	put yanked/copied text after/before cursor
:q	quit file
r	replace character
:set nu	set numbers down left margin
set wm=x	insert newline ’x’ distance from right margin
:sh; shell_type CTRL-D	access a shell
w, W	forward by a word
:w	save file
:wq	save and quit file
x	delete character
yw	yank/copy a word
yy	yank/copy a line
ZZ	save and quit file
xp	transpose two letters

Appendix C. The LART Licence

1998-2000 Jan-Derk Bakker, TU Delft and others™.

C.1. DEFINITIONS

HARDWARE INFORMATION shall mean the CAD databases, schematics, logic equations, bills-of-materials, manufacturing and assembly information, documentation, and any associated information included in this archive.

DEVICE shall mean a physical object based on all or part of the HARDWARE INFORMATION.

C.2. LICENCE

This HARDWARE INFORMATION is copyrighted by Jan-Derk Bakker, TU Delft and other parties. The following terms apply to all files associated with the HARDWARE INFORMATION unless explicitly disclaimed in individual files.

The authors hereby grant non-exclusive permission to use, copy, modify, distribute, and license this HARDWARE INFORMATION, and to build, sell or otherwise distribute an unlimited number of DEVICES for any purpose, provided that existing copyright notices are retained in all copies and that this notice is included verbatim in any distributions. No written agreement, licence, or royalty fee is required for any of the authorized uses. Modifications to this HARDWARE INFORMATION may be copyrighted by their authors and need not follow the licensing terms described here, provided that the new terms are clearly indicated on the first page of each file where they apply.

IN NO EVENT SHALL THE AUTHORS OR DISTRIBUTORS BE LIABLE TO ANY PARTY FOR DIRECT, INDIRECT, SPECIAL, INCIDENTAL, OR CONSEQUENTIAL DAMAGES ARISING OUT OF THE USE OF THIS HARDWARE INFORMATION OR ANY DERIVATIVES THEREOF, EVEN IF THE AUTHORS HAVE BEEN ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.

THE AUTHORS AND DISTRIBUTORS SPECIFICALLY DISCLAIM ANY WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE, AND NON-INFRINGEMENT. THIS HARDWARE INFORMATION IS PROVIDED ON AN "AS IS" BASIS, AND THE AUTHORS AND

DISTRIBUTORS HAVE NO OBLIGATION TO PROVIDE MAINTENANCE, SUPPORT, UPDATES, ENHANCEMENTS, OR MODIFICATIONS.

Glossary

ABI

Application Binary Interface.

AfterStep

A window manager which both emulates and builds upon NEXTSTEP(tm). Features include configurable desktops and the option to use a range of utilities via a series of modules. AfterStep is GNOME-compliant and works with KDE.

ARMLinux

A version of the GNU/Linux operating system targeted at ARM processors.

Assabet

SA-1110 development platform which is suited to the production of handheld and palm devices.

Debian

A Linux distribution which is made up of a kernel, basic tools and applications software. Debian is distinctive because it is produced entirely by volunteers who share a commitment to the development and dissemination of free software. The name derives from the people who founded the Debian Project: Debra and Ian Murdock.

Emacs

An extensible editor with modes for programming, writing and the creation of simple drawings. Emacs has many of the characteristics of a self-contained working environment.

ext2fs

The normal Linux filesystem

fvwm

A window manager which is popular amongst memory-conscious users. Features included extensibility via a series of modules, support for virtual desktops and a 3D-look.

GIMP

An image creation and photo retouching package - extremely powerful. Application currently confined to Linux although ports to Windows and OS/2 are ongoing.

GNOME

GNOME or the GNU Object Model Environment is best regarded as a desktop rather than as a window manager. GNOME will work with any window manager which is GNOME-compliant but the Enlightenment w.m. seems to be the default choice.

GNU/Linux

A synonym for the Linux system. In fact, GNU/Linux is the more accurate term because it makes a distinction between the kernel - Linux - and much of the software which was developed by the GNU Project in association with The Free Software Foundation.

GUI

Graphical user interface. The typical 'desktop' as used by RISC OS, Windows, the Mac and so on. Approximately synonymous with the older term WIMP.

Host machine

When working with an embedded system, this is the desktop machine that contains the development environment. Files are uploaded from this machine to the target device.

IDE

A type of hard drive interface. Provided as standard on Risc PC motherboards. It can have up to two physical drives, configured as ’master’ and ’slave’.

Info

A form of hypertext help page which can be read from within Emacs or by using the info command. These pages can also be printed. Info pages are created with TeXinfo.

KDE

KDE or the K Desktop Environment allows for easy navigation with the aid of the K File Manager, Virtual Desktops and use of KDE-based applications like K-Edit. Currently, installed by default on many Linux distributions with notable exceptions of RedHat (GNOME) and Debian (Window Maker).

LAN server

Any server running on a LAN (Local Area Network). Normally used for file or printer sharing. RISC OS uses Acorn Access. Linux has many, including SAMBA, which is equivalent to an NT or Windows 9x server, NFS (Network File System), and LPD (Printer sharing).

LART

The Linux Advanced Radio Terminal (LART) is a compact energy efficient, embedded computer with a standard configuration of 32MB DRAM and 4MB Flash ROM. Originally developed at the University of Delft.

less

A command which is used to view the contents of a file one screen at a time. Similar commands include cat, more and tail.

Linux Documentation Project LDP

A collaborative project which aims to produce free high quality documentation about GNU/Linux.

MySQL

A Database Management System which is available for both Linux and Windows.

NFS

Network Filing System. The standard UNIX system for sharing disks over the network.

OpenOffice

An office-productivity suite which is approximately comparable to MS Office. It was previously known as StarOffice before it became an open-source project. OpenOffice comes with OpenWriter - a word processor, OpenCalc - a spreadsheet and OpenImpress - a presentational package. OpenOffice requires the X-Windows System although a version exists for Windows.

Open Source

This is a less-confusing name for what is also called ’Free Software’. It describes the development method used for many pieces of software, including the Linux kernel, where the source is freely available for anyone to work on, or modify, or learn from, or use in other projects.

Perl

A powerful interpreted language, which is particularly good for text-processing. Also often used for Web CGI scripts. Famous for being ’powerful but cryptic’. Python is very similar but less cryptic.

Pipe

A way of connecting two programs together so that the output of one program becomes the input of another.

PLEB

An umbrella term for a number of hardware and software initiatives which include the development of a credit-card sized StrongARM prototype for embedded systems and Catapult - a replacement bootloader for Blob.

SCSI

A type of peripheral interface, allowing up to 7 (or even 15) devices. Usually used for scanners, external drives, expansion drives and CDs.

Shell scripts

The UNIX shell processes user commands. It has extensive scripting functions which let you write small programs to automate tasks. These are called scripts. A shell script is equivalent to a RISC OS Obey file or a DOS Batch file, but much more flexible.

Single-tasking

RISC OS normally runs in the familiar desktop. Some utilities (eg hard drive formatters) run ’outside’ the desktop, disabling the normal multi-tasking that goes on. Thus this is referred to as ’single-tasking’ mode. All other RISC OS tasks are suspended until the utility quits.

Swap file or swap partition

A file or partition used by the kernel to allow the machine to appear to have much more memory than it has.

Target device

When working with an embedded device and a separate development system, this is the name for the embedded device. Normally the work is done on the

development system or host' machine.

TeX

A typesetting system which is used to create highly structured documents which may contain alot of mathematics. A number of popular TeX macro packages exist but the most notable is LaTeX.

Texinfo

A documentation system which can produce both on-line and printed documentation from one source file. The source file may be processed by TeX or Groff to create printed output and by makeinfo and Emacs to create an info file.

Vi or Visual Editor

A standard text-editor which is available for nearly every type of UNIX system.

Vim or Vi Improved

A modified version of Vi with additional features. Easy to use for Vi veterans.

Web server

Software that sends web site pages back to browsers. Also referred to as an HTTP daemon (HTTP being the protocol used for web pages). If you run this software then you have set up your own web site.

Window Maker

A window manager which offers mutiple workspaces, graphical configuration tools, pinnable menus and drag-and-drop support. The default window manager for Debian ARMLinux.

Window manager

An X-Windows System program that determines how ’X’ deals with your desktop and client windows.

X-Windows System

Linux’s graphical interface which is commonly know as X11 or more simply as ’X’.